ضمیمه دو

برنامه نویسی تجاری و چند لایه در .NET

در معرفی **.NET**، می‌توان گفت که **.NET** شامل یک مجموعه جدید از تکنولوژی‌ها و نیز یک نمونه جدید برای طراحی و توسعه نرم افزار می‌شود. **.NET** فقط یک محیط طراحی و توسعه جدید نیست، بلکه یک دنباله‌ی کامل از سرورها و سرویس‌ها است که برای حل مشکلات تجاری امروزه به صورت موازی با یکدیگر کار می‌کنند. در طی فصول این کتاب، با برنامه نویسی تحت **.NET** و نیز تعدادی از تکنولوژی‌هایی که در آن به کار رفته است آشنا شدیم و آنها را به طور عملی مورد استفاده قرار دادیم. اما در این ضمیمه سعی می‌کنیم با نگرشی متفاوت **.NET** را بررسی کرده و بعد از معرفی آن، دلیل ایجاد و ابداع **.NET** را بررسی کنیم.

به راحتی می‌توان دریافت که پوشش دادن تمام تکنولوژی‌هایی که برای ارائه یک مدل و راه حل در **.NET** مورد نیاز است حتی در یک کتاب نیز قابل گنجایش نیست. **.NET** علاوه بر در بر داشتن نسخه‌های جدید از تکنولوژی‌های قبلی، چندین تکنولوژی جدید را نیز شامل می‌شود. این مجموعه تکنولوژی‌ها عبارتند از **SQL Server**، **Windows XP**، **NET Enterprise Services**، و همچنین تکنولوژی‌های استاندارد مثل **SOAP** و **XML**.

۲۴-۱ چرا .NET ؟

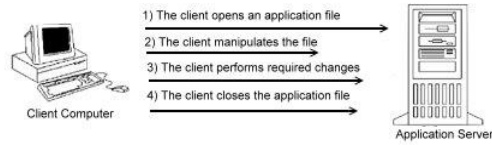
همانند هر تکنولوژی دیگری، **.NET** نیز باید قبل از اینکه مورد استفاده قرار گیرد به صورت کامل بررسی شود. بنابراین در این قسمت سعی می‌کنیم با مزایای استفاده از **.NET**آشنا شویم. در ابتدا نگاه کوتاهی بر مشکلاتی خواهیم داشت که می‌توان با **.NET** به ارائه راه حل برای آنها پرداخت.

۲۴-۲ مشکلات تجاری رفع شده توسط .NET

از بسیاری جهات دنیای کامپیوترهای شخصی با حرکت از حالت محاسبه به صورت مجزا و نیز ایستگاه‌های کاری غیر مرتبط به سوی شبکه‌های کامپیوتری و نیز رابطه‌های سرویس گیرنده/ سرویس دهنده دچار تغییراتی شده است. شبکه‌های سرویس دهنده فایل و یا چاپگر راهی را برای به اشتراک گذاری اطلاعات با مدیریت مرکزی فراهم می‌کند. تولد سیستم‌های سرویس گیرنده/سرویس دهنده به کاهش حجم کار از روی سیستم‌های سرویس گیرنده و انتقال آن به سرورها، و نیز افزایش کارایی و قابلیت اعتماد در برنامه‌ها کمک قابل توجهی کرد. در این نوع برنامه‌ها مدیران سیستم نمی‌توانستند به

کامپیوترهای سرویس گیرنده برای مدیریت و اداره فایلهای موجود اعتماد کنند، زیرا ممکن بود این کامپیوترها به هر دلیلی از کار بایستند و باعث خرابی داده‌های موجود شوند. بنابراین با استفاده از این سرویس‌ها این اطمینان ایجاد می‌شد که سرویس گیرنده محدود به دریافت و مشاهده‌ی فایلها و استفاده از آنها است و سرورها کارهای اصلی را انجام می‌دهند. این مورد باعث افزایش قابلیت اعتماد به برنامه‌ها می‌شد، زیرا احتمال رخ دادن خرابی در آنها به شدت کاهش پیدا می‌کرد.

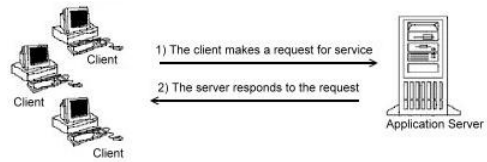
اولین شبکه‌های کامپیوتری شامل سرورهای فایل و چاپگر به همراه یک سیستم متمرکز برای اشتراک اطلاعات و نیز مدیریت شبکه بود (همانند شکل زیر). اما این مدل فقط شامل اشتراک فایل و چاپگر از طریق شبکه بود. برنامه‌ی اصلی در حقیقت به صورت کامل در قسمت سرویس گیرنده عمل کرد و فقط از مزایای مشخصی از سرویس‌های موجود در مدل سرویس گیرنده/سرویس دهنده استفاده می‌کرد.



شکل (۱-۲۴) مدل قدیمی‌سرویس گیرنده/سرویس دهنده – در این مدل فقط داده‌ها از سرور دریافت می‌شد، اما برنامه‌ی اصلی به وسیله سرویس گیرنده کنترل می‌شد

مشکلات این نوع سیستم‌های متمرکز شامل مواردی از قبیل فقدان کارایی و یا خرابی و از دست رفتن اطلاعات می‌شوند. فقدان کارایی به این دلیل بود که برنامه‌های سمت کاربر بایستی تمامی‌محاسبات مورد نیاز را انجام می‌دادند. به علت ریسک ثابت و همیشه موجود نا پایداری سرویس گیرنده[188]، پتانسیل خرابی اطلاعات نیز بالا بود. اگر موقع دستکاری فایل بر روی سرور ارتباط سرویس گیرنده از شبکه قطع می‌شد برنامه یا اطلاعات فایل به راحتی تخریب می‌شدند.

بنابراین مدل قدیمی‌را تصحیح کردند و مدلی مانند شکل زیر را ارائه دادند. مدل تصحیح شده‌ی سرویس گیرنده/سرویس دهنده هرگز به کاربر اجازه نمی‌داد که به صورت واقعی به اطلاعات یا برنامه‌ها دسترسی داشته باشد. سرویس گیرنده هیچ تماسی با فایل یا برنامه‌ی مورد استفاده نداشت و این موجب کاهش قابل توجه ریسک از دست رفتن اطلاعات می‌شد.



شکل (۲-۲۴) این شکل مدل تصحیح شده‌ی سرویس گیرنده/سرویس دهنده را نمایش می‌دهد

[188] ممکن بود به هر دلیلی سرویس گیرنده ای که کاربر از آن استفاده می کند از شبکه قطع شده و یا متوقف شود.

امروزه یک سرور تمامی‌سرویسهای اینترنتی مربوط به برنامه از احراز هویت کاربر تا دسترسی به داده‌ها را انجام می‌دهد. این امر مشکلات قابل توجهی را در مقیاس‌های کوچک ایجاد نمی‌کند، اما وقتی فراوانی درخواست داده‌ها از توانایی سرور بانک اطلاعاتی تجاوز کرد و سرور برنامه تقاضاهایی بیشتر از آنچه توانایی پاسخ گویی به آنها دارد را دریافت کرد، مشکلات عمده‌ای ایجاد می‌شود. پس این مدل نیز دارای ضعف‌هایی است که در مدلهای بعدی رفع خواهند شد.

و اما در مورد قابلیت اطمینان[۱۸۹]. اگر هر بخشی از برنامه که در بیشتر حالتها بر روی سرور قرار می‌گیرد دچار نقص شود، برنامه از کار افتاده و برنامه‌هایی که در حال کار با برنامه مذکور هستند آسیب می‌بینند. پس این مدل از نظر پایداری نیز دارای مشکلاتی است.

مجموعه تکنولوژی‌هایی که در **NET.** وجود دارند شامل تمام عوامل مورد نیاز برای پیاده سازی راه حلهای تجاری، از ابزارهای طراحی گرفته تا سرویسهای تحت وب می‌شود. این مجموعه که باعث افزایش پایداری، قابلیت اعتماد ، انعطاف پذیری و قدرت مدیریت می‌شود، تمام مشکلات مذکور در مدل قبلی را رفع می‌کند. برای نمونه **NET.** سرورها را از این محدودیت که فقط با کاربران در ارتباط باشند رها می‌کند و به آنها اجازه می‌دهد که علاوه بر ارتباط با کاربران، با دیگر سرورها نیز در پشت پرده ارتباط داشته باشد. تعامل سرورها از طریق وب سرویس‌های ارائه شده، نمونه خوبی از این زمینه می‌باشند.

شکل (۳-۲۴) سروری که در کنار کار با سرویس گیرنده با دیگر وب سرویس‌ها نیز تعامل دارد و از آنها استفاده می‌کند

۱-۲-۲۴ کارایی و مقیاس پذیری

اگر یک سیستم قدرت مقیاس پذیری[۱۹۰] نداشته باشد، هیچ راهی برای تهیه برنامه‌هایی که به تعداد زیادی کاربر به طور همزمان پاسخ دهد نخواهد بود. در حقیقت، در سیستم‌های سازمان مقیاس هر چه کارایی سیستم مهم باشد، پایداری و مقیاس پذیری آن از اهمیت بیشتری برخوردار خواهد بود.

---

[189] **Reliability**
[190] **Scalability**

در تعریف کارایی[191] سیستم می‌توان گفت که این عبارت به تعداد سیکل‌هایی که یک پردازنده نیاز دارد تا اجرای یک متد را به پایان برساند اطلاق می‌شود. اما منظور از مقیاس پذیری تعداد کاربرانی است که می‌توانند به صورت همزمان یک وظیفه خاص را در سرور اجرا کنند. برای مثال تصور کنید که یک متد از برنامه، اطلاعات خواسته شده را با سرعتی شگفت انگیز باز میگرداند. ارزش این متد به علت به کارگیری صد درصد پردازنده است. بنابراین در حالی که کارایی چنین سیستمی‌خوب است اما مقیاس پذیری آن ضعیف است، زیرا پردازنده تنها یک یا دو کاربر همزمان که درخواست اجرای متد مذکور را دارند را می‌تواند پشتیبانی کند. مقیاس پذیری یک سیستم کاملاً به معماری درونی برنامه وابسته است.

۳-۲۴ مزایای .NET

برای توسعه دهندگان پاسخ به سوال "چرا **.NET**؟" در دو گروه مزایای موجود در **IDE**[192] مربوط به **.NET** و نیز مفهوم **.NET** قرار می‌گیرد.

یک سری از تکنولوژی‌های موجود در **.NET** شامل مجموعه‌ای از تکنولوژیهایی است که در برقراری ارتباط بین پلت فرم‌های گوناگون مورد استفاده قرار می‌گیرد مانند پروتکلهای استانداردی مثل **HTTP**، و نیز تکنولوژیهایی که به پلت فرم خاصی وابسته نیستند مثل **XML**. این تکنولوژیها باعث می‌شود که سیستم‌های ایجاد شده با **.NET** بتوانند با سیستم‌های قبلی مانند **COM**[193] و **CORBA**[194] از طریق وب تعامل داشته باشند. به علاوه به علت اینکه توجه به پلت فرمهای مقصد نیز هم اکنون رفع شده است، بنابراین برنامه نویسان می‌توانند از آن لحاظ نگرانی نداشته باشند و روی نیازهای تجاری برنامه تمرکز کنند.

البته باید توجه داشته باشید که **.NET** یک هدف در حال تغییر است. اگر چه بسیاری از مواردی که امروزه ما از این تکنولوژی میدانیم تغییر نخواهد کرد، اما همانطور که روز به روز تکنولوژیهای جدید تری در حال توسعه هستند، به یقین این موارد به **.NET** اضافه شده و یا مواردی از آن حذف خواهند شد.

قبل از پیدایش اینترنت بیشتر برنامه‌ها مبتنی بر چند فرم ویندوزی و بدون ارتباط با جهان خارج بود. اما با پیدایش اینترنت شاهد گسترش روزافزون نرم افزارهای تحت وب و نیز تحت شبکه هستیم که موجب تغییرات بسیاری در دنیای امروزه شده‌اند.

---

[191] **Performance**
[192] **Integrated Development Environment**
[193] **Component Object Model**
[194] **Common Object Request Broker Architecture**

در گذشته بیشتر سایتهای وب شامل صفحات ایستا بودند که با ارائه‌ی چندین صفحه اطلاعات که به یکدیگر لینک شده بودند، نیازهای کاربران را برطرف می‌کردند. اما با گسترش روز افزون اینترنت کاربران نیاز به صفحات پویایی که اطلاعات خاص خودشان را نمایش دهد را روز به روز بیشتر احساس می‌کردند.

۱-۳-۲۴ پذیرش استانداردهای همگانی

یکی ار مهمترین جنبه‌های **NET.** پذیرش استانداردهای صنعتی همگانی توسط مایکروسافت است که **XML** یکی از مهمترین آنها است. تصور برخی از افراد از این مورد این است که **XML** پیشرفته ترین تکنولوژی عصر حاضر است، که قطعا این امر برداشتی نادرست است. اما **XML** یکی از بهترین راههای موجود برای یکپارچه سازی سیستمهای نا متجانس محسوب می‌شود.

بزودی تمامی‌سرورهای مایکروسافت به یک سرور **NET.** تبدیل خواهند شد که **XML** و پلت فرم **NET.** را کاملاً پشتیبانی می‌کنند. بنابراین به وسیله پیاده سازی پروتکلهای استاندارد توسط این سرورها، برنامه‌های ارائه شده مبتنی بر سرورهای مایکروسافت قدرت تعامل با پلت فرمهای دیگر را نیز خواهند داشت. در جدول زیر لیستی از سرورهای مایکروسافت و همچنین توضیح کاربرد آنها ذکر شده است.

سرور	شرح
Microsoft Application Center Server 2000	این سرور برنامه‌های مبتنی بر وب را توزیع کرده و همچنین سرورها را به صورت گروهی مدیریت می‌کند.
Microsoft BizTalk Server 2000	پردازش‌های تجاری را پیاده سازی می‌کند و اطلاعات را به وسیله یک رابط استاندارد و پذیرفته شده ارائه می‌دهد.
Microsoft Commerce Server 2000	برای ایجاد برنامه‌های تجارت الکترونیک استفاده می‌شود.
Microsoft Exchange Server 2000	قابلیت انجام مبادلات از طریق اینترنت را فراهم می‌کند.
Microsoft Host Integration 2000	تعامل با کامپیوترهای بزرگ را برقرار می‌کند.
Internet Security And Acceleration 2000	به عنوان یک دیوار آتش عمل می‌کند.
Microsoft SQL Server 2000	سرویسهای تجزیه و تحلیل و نیز نگه داری بانک اطلاعاتی را ارائه می‌دهد.

۲-۳-۲۴ سرویس‌های وب

یکی از عواملی که ممکن است کاملاً جدید به نظر برسد وب سرویس‌ها هستند. وب سرویس‌ها در حقیقت اصولی هستند که زیرساخت بیشتر استراتژی‌های **.NET** را تشکیل می‌دهند و به جرات می‌توان گفت که هدف اصلی از ایجاد **.NET** به شمار می‌روند. وب سرویس‌ها سرویس‌هایی هستند که به وسیله یک برنامه تحت اینترنت ارائه شده و توسط دیگر وب سرویس‌ها یا برنامه‌های سرویس گیرنده استفاده می‌شوند. وب سرویس‌ها برپایه ابزارهای استاندارد مثل **XML** و **HTTP** تولید می‌شوند و مستقل از پلت فرم و محیط تولید آنها می‌باشند. بنابراین برای استفاده از آنها نیازی به **.NET** نیست.
با ترکیب **HTTP** و **XML** و تولید **.NET,SOAP** یک راه حل قابل اعتماد را برای توسعه برنامه‌های تحت وب ارائه می‌دهد. استاندارد **SOAP** که در حقیقت انتقال داده‌های **XML** به وسیله **HTTP** است، پایه و اساس سرویس‌های وب محسوب می‌شود. نه تنها **SOAP** می‌تواند از امکانات **COM** بهرمند شود بلکه قدرت تعامل و استفاده از مزایای استانداردهای دیگری مثل **CORBA** را نیز دارد.

۴-۲۴ ویژگی‌های محیط توسعه Visual Stadio.NET

با وجود اینکه ظاهر **VS.NET** نسبت به قبل تغییرات قابل ملاحظه‌ای کرده است، پیشرفت اصلی این برنامه به علت تکنولوژی‌هایی است که محیط توسعه‌ی **.NET** بر پایه آنها استوار است. در حقیقت این تکنولوژی‌ها است که باعث تولید سریع برنامه همراه با پایداری و قابلیت اعتماد محیط‌های توسعه کلاسیک مانند **C++** می‌شود.

۵-۲۴ Common Language Runtime

یکی از مهمترین پیشرفت‌های ویژوال استودیو اضافه شدن یک محیط مدیریت زمان اجرای مستقل از زبان برنامه نویسی است که **Common Language Runtime** یا **CLR** نامیده می‌شود. **CLR** قدرت طراحی به هر زبانی را در یک محیط مدیریت شده می‌دهد که کمتر دچار نشست حافظه می‌شود و یا با فراهم آوردن **metadata** برای کامپوننت‌ها به آنها اجازه خطا یابی و یا امور دیگر را می‌دهد.
ویژگی‌های کنترل نسخه‌ی برنامه‌ها و یا امنیت آنها در **CLR**، توزیع برنامه را به صورت یک سرویس بسیار ساده تر می‌کند. همچنین **CLR** باعث سادگی و نیز تسریع تولید برنامه‌های تحت وب نسبت به نسخه‌های قبلی می‌شود. درباره این قسمت از **.NET Framework** در ضمیمه بعد صحبت شده است.

۱-۵-۲۴ زبان‌های برنامه نویسی .NET.

با ظهور **.NET** مایکروسافت زبان **C#** را که ترکیبی از قدرت **C++** و راحتی ویژوال بیسیک بود معرفی کرد اما حقیقت در مورد دو زبان اصلی **.NET** بدین صورت است که **C#** و ویژوال بیسیک از لحاظ امکانات بسیار مشابه‌اند.

تمامی‌زبان‌های **NET.** یک زیر مجموعه از امکانات **CLR** را ارائه می‌دهند، اما سطح پشتیبانی آنها از **CLR** متفاوت است[۱۹۵]. برای طراحی بیشتر برنامه‌ها انتخاب زبان تفاوت چندانی ندارد. اما با وجود قابلیت برنامه نویسی با بیش از ۲۰ زبان در **NET.** می‌توان با در نظر گرفتن اینکه بعضی از آنها برای محاسبات علمی بسیار مناسب اند و بعضی دیگر برای پردازش ورودی/خروجی ایده آل هستند زبان مناسبی را برای نوشتن یک برنامه انتخاب کرد.

نکته: *با وجود اینکه **C#** و **VB** کاملا شی گرا هستند اما ممکن است در شرایطی **VC++** کارایی بیشتری را ارائه دهد.*

۲-۵-۲۴ Intermediate Language

این زبان که در **NET.** یک زبان سطح میانی محسوب می‌شود، مستقل از ساختار درونی پردازنده است. تمام کدهای **NET.** در ابتدا به این زبان کامپایل می‌شوند و به علت مستقل بودن این زبان از پردازنده، کامپوننت تولیدی می‌تواند مستقل از پلت فرم عمل کند. از آنجا که **IL** نیز دستورالعمل‌های خود را دارد، کدهای **IL** قبل از اجرا در هر پلت فرمی باید با استفاده از کامپایلرهای مخصوص آن پلت فرم به صورت دستورهای پردازنده‌ی محلی در آید که این امر توسط **JIT** صورت می‌گیرد.

۶-۲۴ تکامل برنامه نویسی لایه‌ای

یکی از مزایای برنامه نویسی مبتنی بر کامپوننت، توانایی برنامه نویس برای تقسیم عملکرد برنامه به چند کامپوننت قابل مدیریت و عمومی است. این امر باعث استفاده مجدد از کد و انعطاف پذیری برنامه می‌شود. همچنین یکی از مزایای اصلی این مدل قدرت تقسیم برنامه به چند لایه موازی است به گونه‌ای که برنامه را بر اساس سرویس‌های آن به چند بخش منطقی تقسیم می‌کند.

۱-۶-۲۴ تعریف

یک **لایه برنامه**[۱۹۶] به یک قسمت از برنامه که به صورت موازی با دیگر قسمت‌ها در حال تعامل است گفته می‌شود. هر لایه‌ی در حال کار در برنامه، وظیفه خاصی را انجام می‌دهد. به عبارت دیگر پیاده سازی برنامه‌های چند لایه عبارت است از تقسیم فیزیکی برنامه به چند قسمت و توزیع هر قسمت بر روی یک سرور.

---

[195] برای اطلاعات بیشتر در این مورد به بخش "خصوصیات عمومی زبانهای برنامه نویسی" در ضمیمه ی بعد مراجعه کنید.

[196] **Application Tier**

مدل برنامه نویسی لایه‌ای در حقیقت برای حل بسیاری از مشکلات برنامه‌های امروزی ایجاد شده است. از این دسته مشکلات می‌توان از مدیریت متمرکز، محاسبات توزیع شده، کارایی برنامه‌ها و مقیاس پذیری آنها نام برد.

۲-۶-۲۴ مدیریت متمرکز

در یک محیط که به صورت متمرکز مدیریت می‌شود، تغییرات موردنظر در یک قسمت مرکزی صورت می‌گیرند و این تغییرات به سرتاسر سیستم توزیع می‌شوند. یک سیستم با مدیریت مرکزی از تعداد نقاط محدودیت قابل مدیریت است.

برای نمونه **MS Application Center 2000**، یک ابزار برای توزیع و مدیریت برنامه‌ها، سیستمی‌است که از مدیریت متمرکز استفاده می‌کند. **Application Center** به علت مدیریت گروهی سرورها باعث افزایش مقیاس پذیری و قابلیت اعتماد برنامه‌ها می‌شود. قابلیت مهمتر این سیستم در این است که اگر یک برنامه در چند سرور اجرا شود و تمام آن سرورها به وسیله‌ی این سیستم به صورت گروهی مدیریت شود، می‌توان از این سیستم برای مدیریت مرکزی برنامه نیز استفاده کرد.

سیستم‌های مدیریت متمرکز، عموماً مدیریت یک گروه از سرورها را به سادگی کنترل یک سرور مستقل انجام می‌دهند و زمانی که یکی از کامپوننت‌های برنامه تغییر کند یا به روز شود، این تغییرات بین تمام سرورهایی که برنامه مذکور را پشتیبانی می‌کنند توزیع می‌شود.

۳-۶-۲۴ محاسبات توزیع شده

در یک محیط محاسبات توزیع شده، پردازش‌ها بین چند سیستم و حتی در صورت نیاز بین چند منطقه تقسیم می‌شود. هدف اصلی این امر افزایش مقیاس پذیری و کارایی شبکه، افزایش قدرت و تحمل نقص است.

۴-۶-۲۴ کارایی

همانطور که ذکر شد منظور از کارایی تعداد سیکل‌های مصرفی برای انجام یک وظیفه خاص است. با وجود اینکه یک وظیفه ممکن است به سرعت انجام شود که این امر حاکی از کارایی قابل قبول برنامه است، اما این کارایی باعث مقیاس پذیری خوب برنامه نمی‌شود. کاربران همیشه کارایی برنامه را در زمان پاسخ گویی آن می‌دانند. زمانی که یک برنامه به خوبی مقیاس پذیر نباشد کاربر تصور می‌کنند که برنامه کارایی خوبی ندارد. بنابراین برای یک برنامه نویس درک تفاوت بین کارایی و مقیاس پذیری از اهمیت بسزایی برخوردار است.

۲۴-۶-۵ مقیاس پذیری

همانطور که ذکر شد منظور از مقیاس پذیری تعداد کاربرانی است که می‌توانند یک وظیفه خاص را به صورت همزمان از سیستم درخواست کنند. نکته اصلی در طراحی برنامه‌های مقیاس پذیر، طراحی کامپوننت‌ها به صورتی است که با کمترین مقدار استفاده از منابع، یک وظیفه خاص را اجرا کند. نتیجه‌ی مطلوب برنامه‌ای خواهد بود که به تعداد موردنظر از کاربران، به صورت همزمان و در یک زمان قابل قبول پاسخ دهد.

۲۴-۶-۶ قواعد و دستورات تجاری

منظور از قواعد تجاری در حقیقت محدودیت‌هایی است که تجارت موردنظر بر روی یک برنامه اعمال می‌کند. کاربرد این قواعد تجاری بر روی جامعیت و یکپارچگی اطلاعات یک برنامه تاثیر می‌گذارد و عدم اجرای کافی این قواعد موجب تاثیرات منفی برنامه می‌شود. با وجود اینکه ممکن است یک راه دقیق برای پیاده سازی این قواعد وجود نداشته باشد، اما همواره راههای قابل قبولی برای اجرای این قواعد به حد کافی وجود دارد.

۲۴-۶-۷ راحتی کاربر

در دنیایی که درک بصری یک واقعیت است، صرف زمان و هزینه بر روی برنامه‌هایی که معماری و قدرت خوبی دارند اما استفاده از آنها مشکل است بی ثمر است. بنابراین حتی اگر یک برنامه کارایی بسیار بالایی داشته باشد و بتواند به طور همزمان به ۱۰۰۰۰۰ کاربر پاسخ دهد، اما کاربران از استفاده از آن برنامه نفرت داشته باشند، نمی‌توان آن را یک برنامه‌ی قابل قبول دانست. با وجود اینکه این مطالب در این قسمت بررسی نمی‌شود، اما محیط‌های طراحی **.NET** ایجاد برنامه‌هایی با ظاهر بسیار زیبا را تا حد ممکن برای برنامه نویسان ساده کرده‌اند.

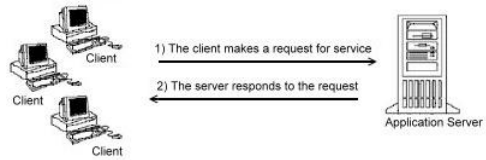
۲۴-۷ برنامه‌های دو لایه

مدل برنامه نویسی دو لایه عموماً به عنوان مدل سرویس گیرنده/ سرویس دهنده یاد می‌شود و این دو واژه با یکدیگر معادل اند. در مدل دو لایه، برنامه‌ی سرویس گیرنده تقاضای اطلاعات یا خدمات را مانند پست الکترونیکی، به یک سرور یا سرویس می‌دهد.

در محیط‌های سرویس گیرنده/ سرویس دهنده معمولا پردازش برنامه‌ها بین سرویس گیرنده و سرویس دهنده تقسیم می‌شود. برنامه سرویس گیرنده رابط کاربر را نمایش می‌دهد و اطلاعات لازم را از کاربر

دریافت می‌کند. برنامه سرور هم معمولاً بر اساس اطلاعات دریافت شده از سرویس گیرنده خدمات مناسبی را ارائه می‌دهد.

شکل زیر یک نمونه از این برنامه‌ها را نشان می‌دهد. با این که مدل نمایش داده شده در شکل کاملاً درست است اما در حالت واقعی تعداد کاربرانی که از یک سرور استفاده می‌کنند بیشتر از چیزی است که در شکل مشاهده می‌کنید.



شکل (۴-۲۴) مدل دو لایه (سرویس گیرنده/سرویس دهنده)

۱-۷-۲۴ مدیریت کد در برنامه‌های دو لایه

دو جنبه‌ی مختلف در مدیریت کد برای محیط‌های سرویس گیرنده/ سرویس دهنده وجود دارد که عبارت اند از مدیریت کد در قسمت سرویس گیرنده و مدیریت کد در قسمت سرویس دهنده.

مدیریت کد در قسمت سرورها عموماً واضح و روشن است. تا زمانی که سرور مورد نظر عضو یک گروه از سرورها نیست، توزیع تغییرات اعمال شده بر روی کد فقط در یک قسمت انجام می‌شود. اما اگر سرور عضو یک گروه از سرورها باشد، تغییرات به وجود آمده در کد باید به صورت دستی یا اتوماتیک به تمام سرورها اعمال شود که معمولا این امر به وسیله نرم افزارهای مدیریتی کلاستری سرورها مانند **MS Application Center 2000** انجام می‌شود.

از سوی دیگر مدیریت کد در بخش سرویس گیرنده عموماً مشکل تر از مدیریت کد در سرورها است. تمام تغییراتی که در برنامه ایجاد می‌شوند باید به تمام کامپیوترهای سرویس گیرنده اعمال شود. چنین توزیع‌هایی که بایستی به صورت همزمان صورت گیرند معمولاً به سطح بالایی از هماهنگی و بر اساس کامپیوترهایی که تغییرات را دریافت می‌کند به یک بسته‌ی قابل توزیع مورد اطمینان نیاز دارند.

توزیع کد در بخش سرویس گیرنده با مشکلات دیگری نیز از قبیل هماهنگی با سرور روبرو است. برای مثال تغییراتی که در برنامه‌ی سرور اعمال می‌شود نیاز به بروز رسانی برنامه‌های سرویس گیرنده دارد و تا زمانی که تمام برنامه‌های سرویس گیرنده بروز نشود برنامه سرور بدون استفاده می‌ماند.

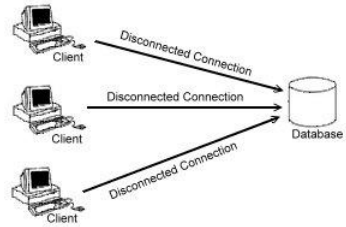
۲-۷-۲۴ کارایی

استفاده از مدل سرویس گیرنده/سرویس دهنده بر روی کارایی شبکه نیز تاثیر می‌گذارد. در این مدل تمام تقاضاهای اطلاعات و خدمات بایستی از طریق شبکه بین سرویس دهنده و سرویس گیرنده جا به

جا شود و این امر باعث محدودیت در شبکه می‌شود. این مشکل زمانی که کاربران زیادی در حال استفاده از سیستم باشند بیشتر مشهود خواهد بود.
هنگامی که کارایی شبکه نزول کند، تقاضاهای دریافتی برای خدمات صف می‌شوند و وقتی تقاضاهای در حال انتظار افزایش یابد سیستم از کار می‌افتد. در این حال تنها راه حل قطع کردن تمام کاربران از شبکه، بازگرداندن تمام تراکنش‌ها به حالت قبلی و در بعضی مواقع راه اندازی مجدد سرور است.
برحسب پروتکل مورد استفاده در شبکه، توزیع برنامه‌های سرویس گیرنده/سرویس دهنده در سطح جهانی غیر ممکن است. زیرا ممکن است برنامه‌ی سرویس گیرنده نیاز داشته باشد در همان شبکه‌ای قرار گیرد که برنامه سرور قرار گرفته است.

۲۴-۷-۳ دسترسی داده‌ها

هنگام ارزیابی کارایی، دسترسی به داده‌ها نیز باید مدنظر قرار گیرند. نه تنها بیشتر برنامه‌های سرویس گیرنده نمی‌دانند که چگونه به سرور بانک اطلاعاتی وصل شوند، بلکه بیشتر آنها احتیاج به یک ارتباط اختصاصی با بانک اطلاعاتی دارند. این مورد که بازای تمام کاربران در حال ارتباط با سرور بانک اطلاعاتی باید یک اتصال فعال برقرار باشد، چه از نظر هزینه پرداختی برای ارتباط با سرور چه از نظر مدیریت مربوط به اتصالات پرهزینه است. همچنین این اتصالات قابلیت به اشتراک گذاشته شدن بین برنامه‌ها را ندارند که این امر تاثیر منفی بر مقیاس پذیری برنامه دارد.



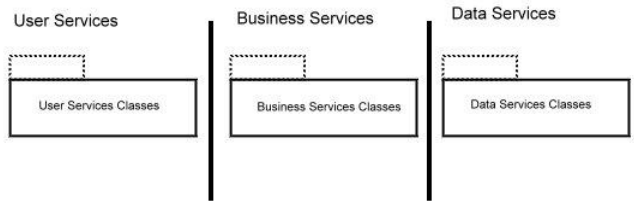
شکل (۵-۲۴) هر سه سرویس گیرنده به اتصال اختصاصی به بانک اطلاعاتی نیاز دارند.

۲۴-۷-۴ قواعد تجاری

در مدل دو لایه یا سرویس گیرنده/سرویس دهنده فقط دو قسمت برای ایجاد و اجرای قواعد تجاری وجود دارد: قسمت سرویس گیرنده و قسمت سرور که خدمات بانک اطلاعاتی را ارائه می‌دهد. البته اجرای این قواعد در سمت سرور منطقی تر است زیرا این امر امکان اینکه یک سرویس گیرنده تغییرات جدید را دریافت نکند را رفع می‌کند.

۸-۲۴ برنامه‌های سه لایه

مدل برنامه نویسی سه لایه با تقسیم برنامه‌های دو لایه به سه قسمت: سرویس‌های کاربران، سرویس‌های تجاری و سرویس‌های اطلاعاتی باعث بهبود برنامه‌های سرویس گیرنده/سرویس دهنده می‌شود. این تقسیم موجب افزایش مقیاس پذیری و قابلیت اعتماد برنامه می‌شود.



شکل (۶-۲۴) مدل منطقی معماری سه لایه

۱-۸-۲۴ سرویس‌های کاربران

لایه سرویس‌های کاربران که همچنین لایه ارائه دهنده[۱۹۷] نیز نامیده می‌شود از فایل‌های اجرایی ویندوز یا صفحات وب مثل **HTML**های دینامیک و یا صفحات **ASP** تشکیل شده‌اند. لایه خدمات کاربران در حقیقت لایه‌ای است که اطلاعات سرور را به کاربر نمایش می‌دهد و اطلاعات مورد نیاز سرور را از کاربر دریافت می‌کند. در مدل سه لایه، لایه ارائه دهنده به دانستن ساختار بانک اطلاعاتی و یا هر سرویس دیگری که به وسیله لایه سرویس‌های اطلاعاتی ارائه می‌شود احتیاج ندارد.

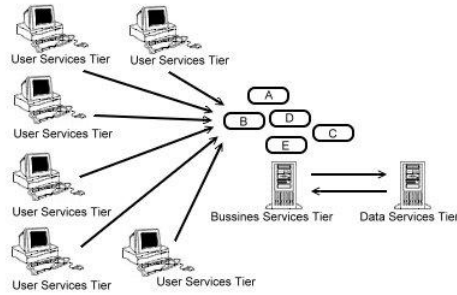
۲-۸-۲۴ سرویس‌های تجاری

لایه دوم، لایه سرویس‌های تجاری است. در برنامه‌های سه لایه، در حقیقت این لایه مسئول چگونگی دسترسی به اطلاعات است. این مسئولیت شامل دریافت تقاضای اطلاعات موردنیاز از طرف لایه سرویس‌های کاربر، فرستادن آن به لایه‌ی داده‌ای و بازگرداندن نتایج درخواست‌ها به کاربر است. این لایه می‌تواند، و در بیشتر شرایط باید، قواعد تجاری اعمال شده در برنامه را نگهداری و اجرا کند.

لایه سرویس‌های تجاری قادر به انجام تمام مواردی است که لایه ارائه دهنده درخواست می‌کند. با وجود اینکه یکی از مهمترین اهداف این لایه، جداسازی لایه ارائه دهنده از لایه سرویس‌های اطلاعاتی است اما این لایه کارهای بسیار بیشتری را انجام می‌دهد. تمام توابع، اعم از محاسباتی و یا وظایف ویژه برنامه، بوسیله این لایه ارائه می‌شوند. تمام کاربران به واسطه لایه ارائه دهنده به امکانات این لایه دسترسی دارند. این لایه به صورت فیزیکی در یکی از سرورهای موجود در شبکه قرار می‌گیرد. تمام

[197] **Presentation Layer**

قواعد تجاری جدید و یا تغییرات اعمال شده در آنها فقط باید در این لایه توزیع شوند، که این امر موجب حذف نیاز به انتشار این قواعد بین کامپیوترهای سرویس گیرنده می‌شود.



شکل (۷-۲۴) کامپوننت‌های لایه سرویسهای تجاری به وسیله تمام سرویسهای کاربر استفاده میشوند

۳-۸-۲۴ سرویس‌های اطلاعاتی

لایه سرویس‌های اطلاعاتی امکان دسترسی به اطلاعات را برای لایه سرویس‌های تجاری فراهم می‌کند که این لایه نیز اطلاعات را در اختیار کاربران سرویس گیرنده در لایه سرویس‌های کاربر قرار می‌دهد.**ADO.NET** و سیستم مدیریت بانک اطلاعاتی**(DBMS)** هر دو در این لایه نگهداری می‌شوند. **ADO.NET** راه حل مایکروسافت برای دسترسی اطلاعات در شبکه و **MS SQL Server** نیز راه حل مایکروسافت برای سیستم‌های مدیریت بانک اطلاعاتی است. هر دو این موارد دسترسی به اطلاعات را فراهم می‌کند. **ADO.NET** روشی را برای دریافت اطلاعات ارائه می‌دهد و **SQL Server** موتور بانک اطلاعاتی است که برای نگهداری خود اطلاعات به کار می‌رود.

۴-۸-۲۴ مدیریت کد

مدیریت کد در برنامه‌های سه لایه بسیار راحت تر از برنامه‌های دو لایه است و دچار مشکلات کمتری می‌شود. به دلیل جداسازی منطقی و فیزیکی برنامه دیگر نیازی به یک تیم توسعه برای کل برنامه نیست. طراحان لایه ارائه دهنده می‌توانند بدون دسترسی به بانک اطلاعاتی به طراحی این لایه بپردازند. برنامه نویسان لایه سرویس‌های تجاری می‌تواند از درگیری با لایه ارائه دهنده آزاد شوند و برنامه نویسان بانک اطلاعاتی می‌توانند بر روی رابطه اطلاعات و یا پیاده سازی قواعد تجاری تمرکز کنند. بواسطه این امر که این لایه‌ها به طور منطقی مجزا از یکدیگر هستند، هر کدام از آنها می‌توانند به طور مجزا کامپایل شده و یا مجدداً پیکر بندی شوند بدون اینکه بر لایه‌های دیگر تاثیر بگذارند.

۲۴-۸-۵ مقیاس پذیری

در برنامه‌های سه لایه مقیاس پذیری بسیار بهبود می‌یابد، زیرا اتصالات بانک اطلاعاتی می‌تواند بدون آنکه قطع شود از کاربری که به آن نیاز ندارد گرفته شده و برای دیگر کاربران نگه داشته شود. این امر تعداد کاربرانی که می‌توانند از طریق لایه میانی(لایه سرویس‌های تجاری) با بانک اطلاعاتی در تماس باشند را افزایش می‌دهد. همچنین در این نوع معماری پردازش از سمت سرویس گیرنده به لایه سرویس‌های تجاری انتقال می‌یابد. کارایی شبکه به علت ارتباط لایه سرویس‌های تجاری با لایه بانک اطلاعاتی به جای ارتباط لایه سرویس گیرنده با بانک اطلاعاتی نیز افزایش می‌یابد. در کل مقیاس پذیری در این سیستم‌ها به واسطه‌ی دسته بندی لایه سرویس‌های تجاری و سرورهای بانک اطلاعاتی افزایش می‌یابد.

کامپوننت‌های تجاری می‌توانند به وسیله ابزارهای ارائه شده توسط **COM+** و یا**Enterprise Services** در حافظه بارگذاری شوند و تا موقع نیاز در آنجا باقی بمانند. این امر موجب افزایش تعداد کاربرانی می‌شود که لایه سرویس‌های تجاری می‌تواند به آنها پاسخ دهد زیرا زمان مورد نیاز برای بارگذاری کامپوننت‌های لازم حذف می‌شود.

۲۴-۸-۶ قواعد تجاری

در برنامه‌های سه لایه قواعد تجاری نباید در لایه ارائه دهنده(لایه کاربر) اعمال شوند. زیرا در این صورت کاربر، به راحتی می‌تواند از این قواعد عبور کند. علاوه بر این با قرار دادن یک قاعده تجاری در سمت سرویس گیرنده این عمل باید برای تمامی‌کامپیوترهای این لایه تکرار شود.

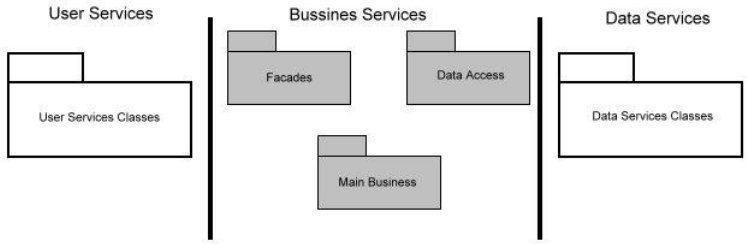
این قواعد می‌توانند در هر یک از لایه‌های سرویس‌های تجاری و یا سرویس‌های اطلاعاتی قرار داده شوند. زمانی که این قواعد در لایه سرویس‌های تجاری قرار می‌گیرند، بایستی اطمینان حاصل شود که کاربران فقط از طریق کامپوننت‌های این لایه به لایه سوم دسترسی دارند و نمی‌توانند از این لایه عبور کرده و به لایه بانک اطلاعاتی دسترسی پیدا کنند. زیرا در غیر این صورت این قواعد به درستی اجرا نمی‌شوند، اما زمانی که این قواعد در لایه بانک اطلاعاتی اجرا شدند دیگر کاربران توانایی عبور از آنها را نخواهند داشت.

زبانهای برنامه نویسی از قبیل**VB.NET، VC++** و یا **C#** برای پیاده سازی این قواعد بهینه شده‌اند. با وجود این، می‌توان با استفاده از امکاناتی که در برنامه‌هایی از قبیل **SQL Server** وجود دارد این قواعد را در لایه بانک اطلاعاتی پیاده سازی کرد.

۹-۲۴ برنامه نویسی چند لایه

تقسیم بندی یک برنامه به چند لایه، به صورت استراتژیک، می‌تواند باعث افزایش مقیاس پذیری، کارایی، انعطاف پذیری و قدرت مدیریت شود، این مورد که به هر لایه یک وظیفه خاص داده شود باعث می‌شود که طراحان آن لایه، بر روی توسعه و پیکر بندی وظیفه مشخص شده عمل کنند.

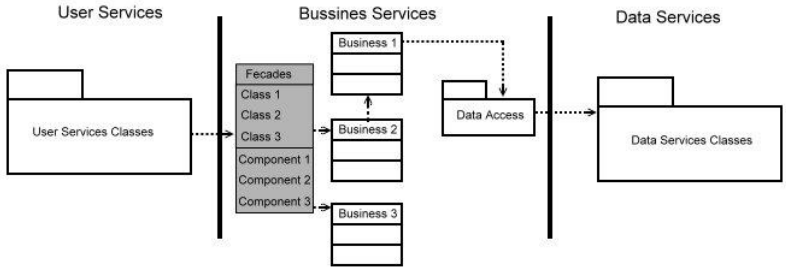
هر برنامه‌ای که شامل بیش از سه لایه شود، جزء برنامه‌های چند لایه قرار می‌گیرد. اما در این بخش منظور از برنامه‌های چند لایه برنامه‌های پنج لایه است. این برنامه‌ها مشابه برنامه‌های سه لایه هستند ولی در آنها لایه سرویس‌های تجاری به سه لایه یا سه کلاس دیگر تقسیم می‌شود که عبارت اند از: کلاس خارجی، کلاس اصلی تجاری و کلاس دسترسی اطلاعات.



شکل (۸-۲۴) کلاسهای لایه تجاری به چند کلاس جدید تقسیم میشوند

۱-۹-۲۴ کلاس خارجی

کلاس خارجی[۱۹۸] در حقیقت به عنوان یک بافر بین لایه سرویس‌های کاربر و امکانات ارائه شده توسط لایه سرویس‌های تجاری برنامه عمل می‌کند. یکی از مزایای استفاده از این کلاس‌ها این است که می‌توان چندین کلاس برای این قسمت تعریف کرد و با استفاده از آنها اطلاعات ایستا را به اطلاعاتی که به سمت سرویس گیرنده فرستاده می‌شوند اضافه کرد و بدین وسیله به طراحان و برنامه نویسان لایه سرویس‌های تجاری در تسریع طراحی و توسعه کمک کرد.



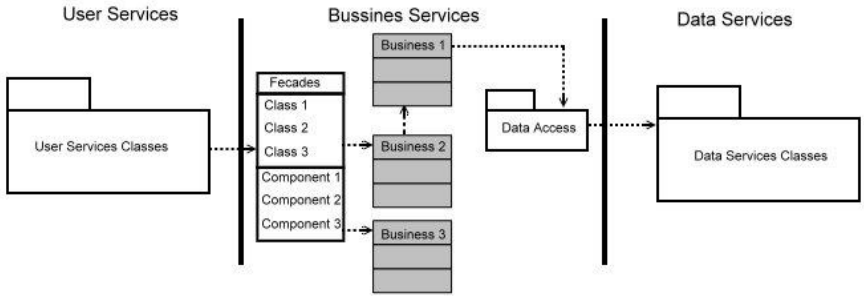
شکل (۹-۲۴) لایه خارجی به عنوان یک بافر برای کامپوننت‌ها عمل می‌کند

[198] Facades Layer

یکی دیگر از مزایای استفاده از این کلاسهای خارجی، توانایی از بین بردن پیچیدگی موجود در توابع لایه سرویس‌های تجاری است. کامپوننت‌های موجود در لایه سرویس‌های تجاری اغلب به گونه‌ای طراحی می‌شوند که بتوانند به طور عمومی به وسیله چند برنامه مورد استفاده قرار گیرند. همچنین هر کاربر و یا سرویسی از سوی صفحات وب، نیاز به نمونه سازی و یا بارگذاری چندین کامپوننت برای اجرای یک وظیفه خاص دارد. به وسیله کلاس‌های خارجی، کاربر لایه سرویس گیرنده فقط نیاز دارد یک نمونه از کلاس خارجی را شبیه سازی کرده و از پیچیدگی کار در لایه میانی جدا می‌ماند.

۲-۹-۲۴ کلاس اصلی تجاری

کلاس اصلی تجاری یا لایه مرحله تجاری[۱۹۹]، در حقیقت فراهم کننده وظایف اصلی تجاری برنامه است که عبارت اند از: اجرای قواعد تجاری، تضمین کردن کارایی منطق تجاری و فراهم کردن دسترسی به کامپوننت‌های اطلاعاتی. به عبارت دیگر این کلاس، هوشمندی واقعی برنامه را ارائه می‌دهد. کلاسهای خارجی نیز با استفاده از کامپوننت‌های این لایه وظیفه موردنظر خود را انجام می‌دهند.



شکل (۱۰-۲۴) رابطه بین کلاس اصلی تجاری با کلاسهای خارجی و کلاسهای دسترسی اطلاعات

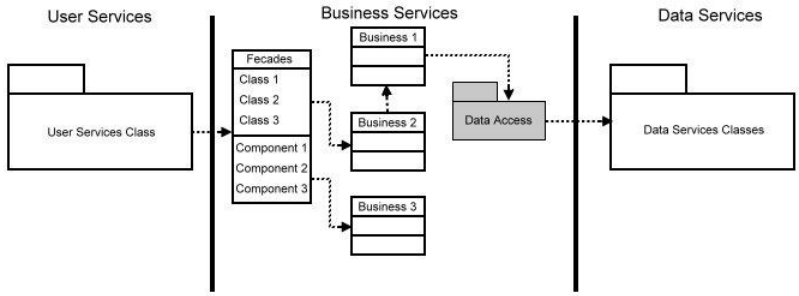
۳-۹-۲۴ کلاسهای دسترسی اطلاعات

کامپوننت‌های دسترسی به اطلاعات یا لایه دسترسی به اطلاعات[۲۰۰] امکان استفاده از اطلاعات را برای لایه اصلی تجاری فراهم می‌کند. در مدل پنج لایه فقط این لایه به ساختار بانک اطلاعاتی دسترسی دارد بنابراین تغییرات در بانک اطلاعاتی فقط نیاز به ایجاد تغییرات در این لایه دارد. قسمت‌های دیگر این برنامه نیازی به اطلاع از زیر ساخت بانک اطلاعاتی ندارند.

---

[199] **Business Level Layer**
[200] **Data Access Layer**

بدون توجه به نوع برنامه استفاده شده در لایه بانک اطلاعاتی، **ADO.NET** روش پیشنهادی مایکروسافت برای این لایه محسوب می‌شود. **ADO.NET** توانایی برقراری ارتباط با انواع منابع اطلاعاتی را از قبیل **SQL Server**، **Oracle**، **Sybase**، **Access**، **Word** ، **Excel** و... را دارد. در طراحی لایه‌ای معمولا منبع اطلاعاتی، یک سیستم مدیریت بانک اطلاعاتی (**DBMS**) است که دسترسی به اطلاعات را فراهم می‌کند. هنگام استفاده از **ADO.NET** طراحان می‌توانند از کامپوننت‌های این کلاس برای ایجاد پرس‌وجوهای مختلف استفاده کنند. علاوه بر این استفاده از روش‌های داخلی سیستم‌های بانک اطلاعاتی نیز امکان پذیر است. مثلا در **SQL Server** می‌توان از پروسیجرهای آماده[۲۰۱] استفاده کرد که حدود ۴۰ در صد سریعتر از روش‌های دیگر است.



شکل (۱۱-۲۴) کامپوننت‌های دسترسی اطلاعات برای دریافت و ارسال اطلاعات از لایه سرویسهای اطلاعاتی به لایه سرویسهای تجاری آماده هستند

۱۰-۲۴ سرویس‌های وب

یک سرویس وب، یک تابع یا مجموعه‌ای از توابع است که از طریق اینترنت قابل دسترسی است و از ترکیب **XML** برای ارائه داده‌ها و **HTTP** برای انتقال داده‌ها بهره می‌برد. وب سرویس‌ها نیز مانند کامپونت‌های **COM** و یا **.NET**. توابع خود را برای کاربران عرضه می‌کنند. استفاده کنندگان این توابع عموماً برنامه‌های سرویس گیرنده هستند. در نتیجه وب سرویس‌ها در طراحی لایه‌ای، به راحتی به عنوان یک لایه الحاقی قابل استفاده‌اند.

وب سرویس‌ها این امکان را به یک توسعه گر می‌دهند که بدون طراحی کامل یک برنامه[۲۰۲]، توابع و وظایف یک برنامه را برای سرویس گیرنده ارائه دهد. یک طراح برنامه‌های سرویس گیرنده می‌تواند در ساخت برنامه خود از امکانات یک یا چند سرویس مبتنی بر وب بهره مند شود.

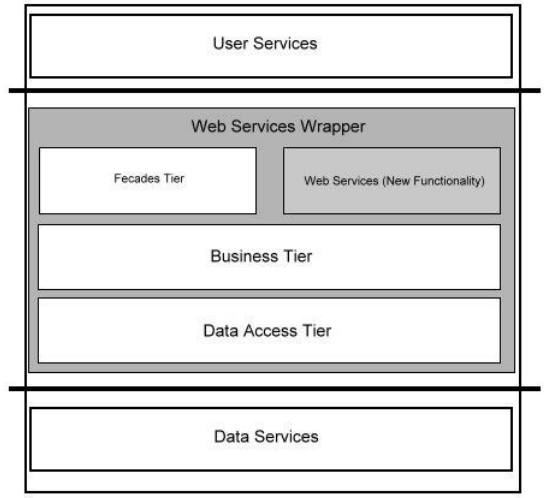
---

[201] **Stored Procedures**

[202] زیرا این برنامه‌ها نیازی به طراحی رابط کاربری ندارند.

۲۴-۱۰-۱ مدل لایه وب سرویس‌ها

همانند لایه خارجی(کلاسهای خارجی که پیشتر توضیح داده شد)، وب سرویس‌ها باعث سادگی استفاده از توابع و نیز کاهش پیچیدگی‌های لایه میانی برای برنامه‌های در حال استفاده می‌شود. به عبارت دیگر برای استفاده از یک وب سرویس در یک برنامه‌ی چند لایه که دارای لایه‌های دسترسی اطلاعات، لایه اصلی تجاری و لایه خارجی است، تنها کافی است که توابع مدنظر از لایه خارجی گرفته شده و توسط وب سرویس عرضه شوند و مابقی توابع توسط لایه خارجی قابل دسترسی باشند. شکل زیر یک مدل منطقی از یک وب سرویس را ارائه می‌دهد که در آن توابعی که توسط لایه خارجی فراهم نشده‌اند، به وسیله این وب سرویس قابل دسترسی هستند.



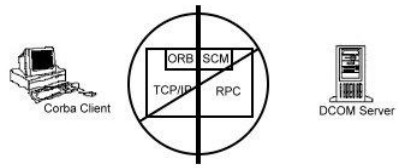
شکل (۲۴-۱۲) یک وب سرویس می‌تواند علاوه بر توابعی که در لایه خارجی ارایه میشوند، توابع مکمل را نیز به لایه سرویس گیرنده ارایه کند

۲۴-۱۰-۲ چرا وب سرویس‌ها؟

یکی از اصلی ترین سوالاتی که در این بخش با آن روبرو هستیم این است که با وجود اینکه می‌توان از تکنولوژی‌های قبلی و راه حل‌های پیشین مایکروسافت همانند **DCOM** برای دسترسی از راه دور استفاده کرد چه نیازی به وب سرویس‌ها است؟

دلیل اصلی این امر مستقل بودن وب سرویس‌ها از پلت فرم است. با وجود اینکه **DCOM** بسیاری از مشکلات توزیع برنامه‌ها و نیز مقیاس پذیری آنها را حل می‌کند، اما وابسته به پلت فرم است. با استفاده از وب سرویس‌ها و تکنولوژی‌های استاندارد صنعتی از قبیل **XML،HTTP** و **SOAP** می‌توان علاوه بر بهره مند شدن از تمام امکانات سیستم‌های قبلی، از قابلیت سازگاری بین پلت فرم‌های گوناگون نیز استفاده کرد.

سیستم‌های قدیمی‌مانند **DCOM،RPC** و **MSMQ** و ... همگی سعی بر این دارند که پلی را برای ارتباطات اینترنتی ایجاد کنند و با وجود اینکه این موارد در داخل پلت فرم‌های مایکروسافت موفق بوده‌اند اما در ارتباط با دیگر پلت فرم‌ها (همانطور که در شکل نشان داده شده است) با مشکلات زیادی روبرو بودند. با استفاده از وب سرویس‌ها سعی شده است که عمده مشکلات این ناهماهنگی رفع شود.



شکل (۱۳-۲۴) ناسازگاری قابلیت استفاده از راه دور بدون پروتکل استاندارد

ضمیمه سه

معماری پلت فرم .NET Framework.

در این ضمیمه سعی شده است که معماری درونی پلت فرم **.NET** بـه صـورتی گـذرا مـورد بررسـی و تحلیل قرار گیرد. هدف از این بخش نگاهی سـطحی بـه معمـاری داخلـی **.NET Framework** و نیـز معرفی تکنولوژی‌هایی است که این پلت فرم دربـر دارد. همچنـین در ایـن بخـش پروسـه تبـدیل یـک سورس کد به یک برنامه قابل توزیع و روش اجرای آن در سیستم عامل مورد بررسی قرار می‌گیرد.

۱-۲۵ کامپایل سورس کد به ماژول‌های مدیریت شده

در طراحی یک برنامه، اولین مرحله تعیین نوع برنامه‌ای است که قصد ایجـاد آن را داریـم، بـرای مثـال برنامه‌ی تحت وب، برنامه‌ی تحت ویندوز، وب سـرویس و یـا انـواع برنامـه‌هـای دیگـری کـه در **.NET** می‌توان ایجاد کرد. فرض می‌کنیم که این قسمت مهم از برنامه به پایان رسیده است و مدل کلی برنامه و ویژگی‌های آن با جزئیات کامل مشخص شده‌اند.

در مرحله‌ی بعد بایستی زبانی که برنامه با آن نوشته خواهد شد انتخـاب شـود. ایـن مرحلـه از اهمیـت بالایی برخوردار است، زیرا زبانهای مختلف امکانات متفاوتی را ارائـه مـی‌دهنـد. بـرای نمونـه در **C** (یـا زبانهای وابسته به آن مانند **C++** و ...) که یک زبان نسبتا سطح پایین است طراح برنامه قدرت کنتـرل کامل بر روی سیستم را دارد، می‌تواند به روش دلخواه حافظه را مدیریت کند و یا به راحتی تردهای[۲۰۳] جدید در برنامه ایجاد کند. از سوی دیگر زبانهایی مثل ویژوال بیسیک، بـه طـراح برنامـه قـدرت ایجـاد سریع رابطهای قوی گرافیکی کاربر را میدهد. به علاوه این زبانها به راحتی می‌توانند با اشـیای **COM** و یا بانکهای اطلاعاتی رابطه برقرار کنند.

البته در **.NET** با تغییراتی که نسبت به سیستم‌های قبلی به وجود آمده است، تا حـد ممکـن زبانهـای مختلف به یکدیگر شبیه شده‌اند و از قابلیتهای نسبتاً یکسانی برخوردار هستند. دلیل این مـورد نیـز در این است که برنامه‌های نوشته شده در این زبانها، هنگام اجرا از یـک محـیط مخـصوص بـه نـام **CLR** استفاده می‌کنند. **CLR** یا **Common Language Runtime** همانطور که از اسم آن نیز مشخص است یک محیط زمان اجرا برای برنامه‌هایی اسـت کـه تحـت **.NET** نوشـته مـی‌شـوند. بنـابراین ویژگیهـا و قابلیتهای موجود در **CLR** برای تمام زبانهایی که از آن استفاده می‌کنند قابل استفاده است. برای مثال

---

[203] **Thread**

اگر **CLR** توانایی ایجاد تردها را داشته باشد، تمام زبانهای برنامه نویسی که از آن استفاده می‌کنند نیز قابلیت ایجاد تردها را دارند. اگر این محیط از **Exception**ها برای پیگیری استثناهای برنامه استفاده کند، تمام زبانها نیز همین روش را دنبال خواهند کرد.

در حقیقت در زمان اجرا این مورد که در طراحی برنامه از چه زبانی استفاده شده است تفاوتی ندارد. بنابراین هنگام انتخاب زبان برنامه نویسی مهمترین موضوعی که باید مدنظر قرار گیرد این است که با استفاده از چه زبانی می‌توان برنامه را ساده تر و سریعتر پیاده سازی کرد.

سوالی که ممکن است در این قسمت ایجاد شود این است که اگر استفاده از زبانهای مختلف تفاوتی ندارد، پس دلیل وجود چند زبان برای برنامه نویسی چیست؟ در پاسخ به این سوال می‌توان گفت که زبانهای مختلف گرامر و سینتکس‌های متفاوتی دارند. اهمیت این مورد را نباید ناچیز در نظر گرفت. برای مثال طراحی یک برنامه برای استفاده در امور اقتصادی، با استفاده از گرامر موجود در زبانهایی مثل **APL** نسبت به **Perl** موجب چندین روز صرفه جویی در زمان پیاده سازی برنامه می‌شود.

مایکروسافت چندین کامپایلر برای استفاده از **CLR** برای زبانهای مختلف ارائه داده است که در مجموعه‌ای به نام **Visual Studio** قرار دارند. این زبانها عبارتند از:

- **Visual C++ with Managed Extensions**
- **C#**
- **Visual Basic**
- **Jscript**
- **J#**
- **IL که یک اسمبلر سطح میانی به شمار می‌رود.**

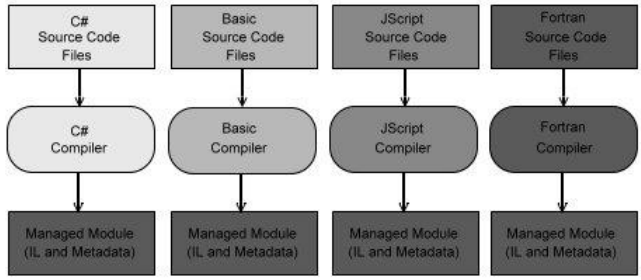
علاوه بر مایکروسافت شرکتهای دیگر نیز برای زبانهای خود کامپایلرهایی را عرضه کرده‌اند که **CLR** را به عنوان محیط زمان اجرای نهایی مورد استفاده قرار داده‌اند. تاکنون کامپایلرهایی برای زبانهای زیر ارایه شده‌اند:**Alice**، **APL**، **Cobol**، **Component Pascal**، **Eiffel**، **Fortran**، **Haskell**، **Mercury**، **ML**، **Mondrian**، **Oberon**، **Perl**، **Python**، **RPG**، **Scheme** و **Smalltalk**.

شکل زیر پروسه‌ی کامپایل فایلهای سورس کد را نمایش میدهد. همانطور که در شکل مشخص است، بدون توجه به کامپایلری که مورد استفاده قرار می‌گیرد خروجی تمام آنها یک ماژول مدیریت شده[۲۰۴]

---

[204] **Managed Module** – منظور کد هایی هستند که در زمان اجرا شدن به وسیله ی **CLR** مدیریت می شوند. در مقابل ماژول های مدیریت شده، ماژول های مدیریت نشده وجود دارند که به صورت عادی و بدون نظارت یک محیط زمان اجرا مانند **CLR**، به وسیله ی سیستم عامل اجرا می شوند. نمونه برنامه های مدیریت شده،

است. ماژول‌های مدیریت شده در حقیقت فایلهای اجرایی[۲۰۵] استاندارد ویندوز هستند که برای اجرا شدن به **CLR** نیاز دارند.



شکل (۱-۲۵) کامپایل سورس کدها به ماژول‌های مدیریت شده

یک ماژول مدیریت شده از قسمتهای مختلفی تشکیل شده است که در جدول زیر توضیح داده شده است:

بخش	توضیح
PE Header	در این هدر مواردی از قبیل نوع فایل (**GUI,CUI,DLL**) و یا تاریخ‌های مربوط به ایجاد فایل، آخرین تغییرات و ... قرار دارند. برای فایلهایی که فقط شامل کد **IL** هستند این قسمت در نظر گرفته نمی‌شود اما برای فایلهایی که شامل کد زبان ماشین هستند این قسمت محتوی اطلاعاتی راجع به کد نیز هست.
CLR header	این قسمت شامل اطلاعاتی است که فایل مورد نظر را به یک ماژول مدیریت شده تبدیل می‌کند (این قسمت از فایل به وسیله **CLR** و یا برنامه‌های وابسته تفسیر می‌شود). این قسمت شامل نسخه **CLR** مورد استفاده برای فایل، مکان و اندازه **metadata** در فایل، منابع، تعدادی فلگ خاص و آدرس ورودی مربوط به فایل آغازین برنامه در **metadata** است.

برنامه هایی است که به صورت عادی به وسیله ی **.NET** ایجاد می شوند. نمونه برنامه های مدیریت نشده نیز، فایلهای اجرایی معمولی هستند که خارج از محیط **.NET** ایجاد شده‌اند.

[205] **Portable Executable Files (PE Files)**

Metadata	هر ماژول مدیریت شده شامل جدول‌هایی است که به نام **metadata** شناخته می‌شوند. دو نوع جدول کلی در این قسمت وجود دارند. نوع اول شامل جداولی است که اطلاعات مربوط به کلاس‌ها و توابع موجود در برنامه را نگهداری می‌کند. نوع دوم شامل جداولی است که محتوی کلاس‌ها و توابع خارجی است که به وسیله این فایل مورد استفاده قرار گرفته است.
کدهای **Intermediate Language (IL)**	کدی است که کامپایلر موقع کامپایل سورس کد تولید می‌کند. موقع اجرای برنامه، **CLR** این کد را به وسیله‌ی **JIT** به کد زبان ماشین تبدیل می‌کند.

بیشتر کامپایلرهای قبلی، کد مربوط به یک معماری خاص از پردازنده را تولید میکردند. برای مثال کدهایی مربوط به **x86**، **IA64**، **Alpha** و یا **PowerPC**. اما کامپایلرهای سازگار با **CLR** هنگام کامپایل سورس برنامه کدهای **IL** تولید می‌کنند. در حقیقت این کدهای **IL** هستند که از آنها به عنوان ماژول‌های مدیریت شده یاد می‌شود، زیرا **CLR** مسئول مدیریت این کدها در زمان اجرا است.
علاوه بر تولید کد **IL**، کامپایلرهای سازگار با **CLR** وظیفه دارند که اطلاعات کامل **metadata** را نیز در فایل قرار دهند. به بیان ساده، **metadata** جدولی درونی در فایل است که توضیحاتی راجع به کلاس‌ها و توابع استفاده شده در برنامه را شامل می‌شود. به علاوه، **metadata** شامل جدولی است که محتوی کلاس‌های خارجی است که برنامه استفاده می‌کند. **Metadata** در حقیقت نسخه جدید تکنولوژی‌هایی مثل **Type Library** و یا فایلهای **IDL**[۲۰۶] است. اما نکته مهم این است که **metadata** نسبت به این تکنولوژی‌ها بسیار کاملتر است و بر خلاف موارد ذکر شده، **metadata**ها در فایلی قرار می‌گیرند که محتوی کد **IL** است. به دلیل اینکه کامپایلر این اطلاعات را هنگام اجرای کد تولید می‌کند، امکان ناهماهنگی بین این اطلاعات و کد اصلی نیز از بین خواهد رفت.
بعضی از موارد استفاده **metadata**عبارت اند از:

☑ ***Metadata** نیاز به فایل‌های هدر و یا کتابخانه‌ها را هنگام کامپایل حذف می‌کند، زیرا تمام اطلاعات مورد نیاز در مورد توابع و کلاسهای استفاده شده در فایل حاوی کد **IL** در خود فایل قرار دارند. کامپایلرها می‌توانند اطلاعات **metadata** را به صورت مستقیم از داخل فایلهای مدیریت شده استخراج کرده و از آنها استفاده کنند.*

[206] **Interface Definition Language**

- ☑ *محیط‌های طراحی از قبیل ویژوال استودیو، از این اطلاعات برای کمک به برنامه نویس هنگام نوشتن کد استفاده می‌کنند. در حقیقت ویژگی هوشیاری در ویژوال استودیو (**IntelliSense**) که برای کامل کردن کدها و نمایش اطلاعات لازم در مورد توابع مورد استفاده در کد به کار می‌رود، با استفاده از تحلیل اطلاعات موجود در **metadata** انجام می‌شود.*
- ☑ *ویژگی بررسی و تایید امنیت کد، با استفاده از اطلاعات موجود در **metadata** صورت می‌گیرد. به کمک این ویژگی **CLR** موجب می‌شود که فقط کدهایی که دارای دستورات تبدیل متغیر به صورت امن (**Safe**) هستند صورت گیرند (**Verification**).*
- ☑ ***Metadata** به یک شیئ اجازه می‌دهد که سریالایز شده و در حافظه قرار گیرد[۲۰۷]، به وسیله شبکه به یک وسیله دیگر منتقل شود و در آنجا مجدداً به حالت اولیه تبدیل شود.*
- ☑ ***Metadata** به **GC**[۲۰۸] این امکان را می‌دهد که طول عمر اشیا را کنترل کند، در صورت لزوم آنها را حذف کرده و حافظه تخصیص داده شده به آنها را آزاد کند. برای تمام انواع اشیا، **GC** می‌تواند نوع شیئ را تشخیص دهد و سپس با استفاده از **metadata** ترتیب اشاره اشیا به یکدیگر را تشخیص دهد.*

کامپایلرهای زبان‌های **C#**، **Visual Basic**، **Jscript**، **J#** و نیز اسمبلر **IL** همواره ماژول‌های مدیریت شده‌ای تولید می‌کنند که برای اجرا به **CLR** احتیاج دارند. همانطور که برای اجرای برنامه‌های **VB 6** و یا برنامه‌های **MFC** در کامپیوتر مقصد، به کتابخانه‌های **Visual Basic** و یا کتابخانه‌های **MFC**[۲۰۹] نیاز است، اجرای فایل‌های مدیریت شده که توسط این کامپایلرها تولید می‌شوند نیز به **CLR** احتیاج دارند.

کامپایلر **C++** مایکروسافت بر خلاف دیگر زبان‌ها، کدهای مدیریت نشده (کدهای عادی **EXE** و یا **DLL** که هم اکنون وجود دارند) تولید می‌کند. این فایلها برای اجرا به **CLR** نیاز ندارند. برای تولید کد مدیریت شده توسط این کامپایلر باید هنگام کامپایل از یکی از سوییچ‌های خط فرمان استفاده کرد. بین تمام کامپایلرهایی که ذکر شد فقط کامپایلر **C++** مایکروسافت این امکان را به برنامه نویس می‌دهد که در یک ماژول از هر دو نوع کد مدیریت شده و مدیریت نشده استفاده کند. این موضوع هنگام طراحی کد این امکان را به برنامه نویس می‌دهد که کلاسهای جدید خود را به صورت مدیریت شده ایجاد کرده، ولی همچنان از کلاسهای مدیریت نشده قبلی نیز استفاده کند.

---

[207] **Serialization**
[208] **Garbage Collector**
[209] **Microsoft Foundation Class**

۲-۲۵ ترکیب ماژول‌های مدیریت شده در اسمبلی‌ها

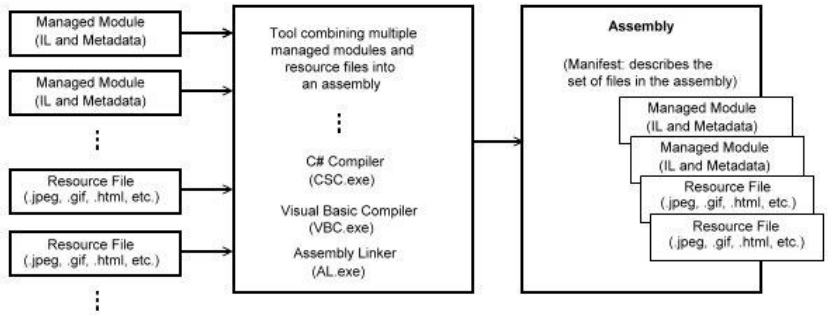
در واقعیت **CLR** با ماژول‌های مدیریت شده کار نمی‌کند، بلکه با اسمبلی‌ها کار می‌کند. اسمبلی، مفهومی‌انتزاعی است که درک آن ممکن است ابتدا مشکل به نظر رسد. در تعریف اول، اسمبلی یک ترکیب منطقی از یک یا چند ماژول مدیریت شده است. در تعریف دوم می‌توان گفت که اسمبلی کوچکترین واحدی است که قابلیت استفاده مجدد[۲۱۰]، تعیین سطوح امنیتی و یا تعیین نسخه[۲۱۱] را دارد. بسته به انتخاب برنامه نویس می‌توان به وسیله ابزارهای مورد استفاده برای کامپایل برنامه، فایلهای اسمبلی یا ماژول‌های مدیریت شده تولید کرد.

شکل زیر به درک مفهوم اسمبلی کمک می‌کند. همانطور که در این شکل مشاهده می‌کنید، بعضی ماژول‌های مدیریت شده و منابع برنامه به وسیله یک ابزار تحت پردازش قرار گرفته‌اند. این ابزار یک فایل اجرایی تولید می‌کند که گروه بندی منطقی چند فایل را بیان می‌کند. فایل تولید شده شامل یک بلاک داده‌ای است که **manifest** نامیده می‌شود. **Manifest** که در واقع به تعدادی از جدول‌های موجود در **metadata** اطلاق می‌شود، شامل اطلاعاتی در رابطه با فایل‌هایی است که اسمبلی را تشکیل می‌دهد. به علاوه اطلاعاتی راجع به توابع و کلاس‌هایی از این اسمبلی که می‌تواند به وسیله‌ی دیگر اسمبلی‌ها مورد استفاده قرار گیرد (مانند کلاسهای **public**)، و نیز منابع و یا فایلهای اطلاعاتی که با این اسمبلی در ارتباط هستند نیز در **manifest** قرار داده می‌شود.

به صورت پیش فرض کامپایلرها هنگام کامپایل، ماژول‌های مدیریت شده را به فایلهای اسمبلی تبدیل می‌کنند. به طور مثال، کامپایلر بیسیک هنگام کامپایل، با اضافه کردن **manifest** به ماژول مدیریت شده و انجام کارهایی از این قبیل یک اسمبلی تولید می‌کند. بنابراین هنگام استفاده از این کامپایلرها، برای تولید یک اسمبلی نیازی به استفاده از سوییچ خاصی نیست. اما در شرایطی که می‌خواهیم با استفاده از چند ماژول مدیریت شده که هر یک در فایل مخصوص به خود قرار دارند، یک اسمبلی ایجاد کنیم باید ابزارهای دیگری مثل لینکرها[۲۱۲] را به کار ببریم.

---

[210] **Reusability**
[211] **Versioning**
[212] **Assembly Linker (al.exe)**



شکل (۲-۲۵) ترکیب ماژول‌های مدیریت شده در اسمبلی‌ها

این امر که یک اسمبلی از چند فایل تشکیل شده و هر کلاس در یک فایل قرار داده شود و یا اینکه تمام کلاسهای موجود در اسمبلی، همه در یک فایل قرار گیرند کاملا به طراح برنامه و برنامه نویس بستگی دارد. به طور مثال تصور کنید که می‌خواهید برنامه‌ی خود را از طریق وب توزیع کنید. به این ترتیب می‌توانیم کلاس‌هایی از اسمبلی مربوط به برنامه که کمتر مورد استفاده قرار می‌گیرد را در یک فایل و کلاسهای مهم و پر کاربرد را در فایل دیگر قرار دهیم. بدین وسیله فایل حاوی قسمتهای غیر ضروری فقط زمانی از اینترنت دریافت می‌شود که به آنها نیاز است. این عمل موجب افزایش سرعت بارگذاری برنامه از وب می‌شود و نیز فضای کمتری را در دیسک اشغال می‌کند.

یکی دیگر از ویژگی‌های یک اسمبلی این است که اطلاعات کافی درباره منابع و اسمبلی‌های دیگری که از آنها استفاده می‌کند (برای مثال شماره نسخه آنها) را نیز در خود نگهداری می‌کند. این امر موجب می‌شود که **CLR** برای اجرای اسمبلی به موارد اضافی نیاز نداشته باشد. به عبارت دیگر اجرای این اسمبلی‌ها به ایجاد تغییرات در رجیستری و یا **Active Directory** ندارد. به همین دلیل توزیع این اسمبلی‌ها بسیار ساده تر از توزیع کامپوننت‌های مدیریت نشده است. خاصیت استفاده از روش **XCOPY** برای توزیع برنامه‌های نوشته شده به وسیله‌ی **.NET** نیز به همین دلیل است که اسمبلی‌های مورد استفاده در یک برنامه می‌توانند به صورت کامل خود و همچنین فایلهای مورد نیازشان را توصیف کنند.

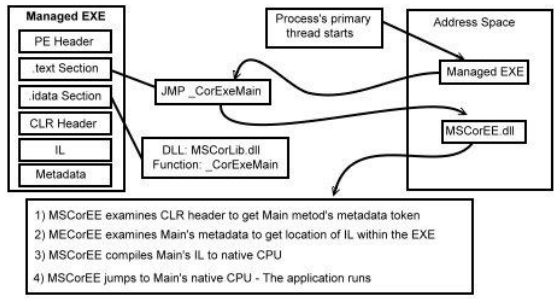
۳-۲۵ بارگذاری Common Language Runtime

هر اسمبلی که ایجاد می‌شود می‌تواند یا یک فایل اجرایی باشد و یا یک فایل **DLL** که شامل چند کلاس برای استفاده توسط فایلهای اجرایی است. در هر صورت **CLR** مسئول اجرای کدهای داخل این اسمبلی است. این امر بدین معنی است که برای اجرای این فایلها باید ابتدا **.NET Framework** در

کامپیوتر مقصد نصب شود. البته در نسخه‌های جدید ویندوز به صورت درونی **.NET Framework** وجود دارد و نیازی به نصب آن نیست.

هنگام ایجاد یک اسمبلی اجرایی (**EXE**)، کامپایلر مقداری اطلاعات خاص را به بخش **.text** در قسمت هدر این فایل اضافه می‌کند. این اطلاعات زمان اجرای برنامه موجب اجرای **CLR** می‌شوند. سپس **CLR** با تشخیص نقطه ورودی برنامه، به برنامه اجازه می‌دهد که آغاز به کار کند.

به طور مشابه اگر یک فایل اجرایی مدیریت نشده، با استفاده از **LoadLibrary** سعی در بارگذاری و استفاده از یک اسمبلی مدیریت شده را داشته باشد، قبل از اجرای فایل برنامه، **CLR** اجرا شده و کنترل فایل مدیریت شده را در دست می‌گیرد.



شکل (۳-۲۵) بارگذاری و مقدار دهی اولیه به **CLR**

هنگامی که کامپایلر یک فایل اجرایی را ایجاد می‌کند، عبارت زیر را به بخش **.text** در هدر مربوط به فایل اضافه می‌کند:

JMP _CorExeMain

این دستور، یک دستور به زبان اسمبلی است که زیر برنامه‌ای به نام **_CorExeMain** را فراخوانی می‌کند. به دلیل اینکه زیر برنامه‌ی **_CorExeMain** در فایل **MSCorEE.dll**[۲۱۳] قرار دارد، پس اسم **MSCorEE.dll** نیز به عنوان یکی از کتابخانه‌های کلاسی که توسط فایل مورد استفاده قرار گرفته است در بخش **.idata** اضافه می‌شود. زمانی که یک فایل اجرایی مدیریت شده فراخوانی می‌شود، ویندوز با آن به صورت یک فایل مدیریت نشده و معمولی برخورد می‌کند. برنامه‌ی مخصوص بارگذاری در ویندوز، با بررسی قسمت **.idata** شروع به بارگذاری فایل **MSCorEE.dll** در حافظه می‌کند و آن را در حافظه قرار می‌دهد. سپس برنامه با بدست آوردن آدرس تابع **_CorExeMain** این متد را اجرا می‌کند.

به این ترتیب پروسه اصلی که مربوط به اجرای برنامه است به اجرای تابع **_CorExeMain** می‌پردازد. اجرای این تابع منجر به اجرا شدن **CLR** و بارگذاری آن در حافظه خواهد شد. با اجرا

[213] **Microsoft Component Object Runtime Execution Engine**

شدن **CLR**، کنترل برنامه در اختیار آن قرار می‌گیرد. **CLR** نیز ابتدا با بررسی بخش هدر **CLR** مربوط به فایل، نقطه آغازین برنامه را مشخص می‌کند. سپس **CLR** کد **IL** مربوط به تابع فراخوانی شده را به وسیله‌ی **JIT** به کد زبان ماشین تبدیل کرده و آن را اجرا می‌کند (این عمل در پروسه اصلی برنامه صورت می‌گیرد). در این مرحله اجرای کد مدیریت شده آغاز می‌شود.

همین شرایط برای ایجاد **DLL**های مدیریت شده نیز صادق است. اگر این **DLL** به وسیله‌ی یک فایل اجرایی که با **NET.** ایجاد شده است مورد استفاده قرار بگیرد، پس حتماً تاکنون **CLR** اجرا شده است و می‌تواند کنترل این **DLL** را نیز در دست بگیرد. اما تصور می‌کنیم که این **DLL** می‌خواهد به وسیله‌ی یک فایل اجرایی عادی، با استفاده از تابع **LoadLibrary** استفاده شود. در این شرایط قبل از اینکه **DLL** اجرا شود، باید **CLR** در حافظه قرار گیرد تا بتواند کدهای موجود در **DLL** را کنترل کند. بنابراین در این شرایط نیز اتفاقاتی همانند اجرا شدن یک فایل **EXE** که با **NET.** ایجاد شده است رخ می‌دهد. یعنی هنگام ساختن **DLL**های مدیریت شده، کامپایلر دستور زیر را به بخش **text.** هدر مربوط به اسمبلی اضافه می‌کند:

```
JMP _CorDllMain
```

زیر برنامه‌ی **_CorDllMain** نیز در فایل **MSCorEE.dll** قرار داده شده است و این امر موجب می‌شود که نام این فایل به بخش **idata.** اضافه شود. هنگامی که ویندوز فایل مربوط به **DLL** را بارگذاری می‌کند، ابتدا فایل **MSCorEE.dll** را در حافظه قرار می‌دهد (البته اگر تاکنون توسط برنامه‌ی دیگری بارگذاری نشده باشد) و سپس آدرس تابع مذکور را در حافظه بدست می‌آورد. پروسه‌ای که تابع **LoadLibrary** را در ابتدا فراخوانی کرده بود تا کدهای درون **DLL** را اجرا کند، تابع **_CorDllMain** را از فایل **MSCorEE.dll** اجرا می‌کند. اجرای این تابع نیز موجب راه‌اندازی **CLR** شده و به این ترتیب کد مدیریت شده می‌تواند به صورت عادی اجرا شود..

مواردی که برای اجرای فایلهای محتوی کد مدیریت شده ذکر شد فقط در ویندوزهای **98**، **98 SE**، **ME**، **NT 4** و **2000** لازم است زیرا این سیستم عاملها قبل از عرضه **CLR** به وجود آمده‌اند. ویندوز **XP** و ویندوز سرور ۲۰۰۳ به صورت درونی اجرای فایلهای مدیریت شده را پشتیبانی می‌کنند و نیازی به موارد ذکر شده نیست. نکته دیگر این است که این توابع مخصوص دستگاههای **x86** هستند و در مدل‌ها و ساختارهای دیگر پردازنده درست عمل نمی‌کنند.

۴-۲۵ اجرای کدهای مدیریت شده

همانطور که ذکر شد فایلهای اسمبلی محتوی **metadata** و کدهای **IL** هستند. **IL** یک زبان سطح میانی است که توسط مایکروسافت ایجاد شده است. این زبان نسبت به زبان ماشین بسیار سطح بالاتر است. تشخیص اشیای ایجاد شده از کلاسها و دستوراتی برای ایجاد و مقدار دهی اولیه آنها، توانایی

فراخوانی توابع داخل اشیا، قابلیت خطا یابی در داخل برنامه‌ها و نگهداری عناصر در یک آرایه به صورت مستقیم از توانایی‌های این زبان محسوب می‌شود. در حقیقت این زبان یک نسخه شیئ گرا از زبان ماشین به شمار می‌رود.

معمولا برنامه نویسان برای طراحی برنامه از زبانهای سطح بالا مانند **C#** و یا **Visual Basic** استفاده می‌کنند. سپس کامپایلر کد این زبانها را به **IL** تبدیل می‌کند.

بعضی از برنامه نویسان عقیده دارند که زبان **IL** نمی‌تواند به طور کامل از الگوریتم‌هایی که آنها استفاده کرده‌اند محافظت کند. به عبارت دیگر، به نظر آنها می‌توان یک فایل حاوی کد مدیریت شده تولید کرده و سپس با استفاده از برنامه‌هایی که می‌توانند کد **IL** را نمایش دهند (مانند **ildasm**[۲۱۴]) و مشاهده کد **IL** یک برنامه، به الگوریتم اصلی آن دست پیدا کنند.

این امر کاملاً درست است و برنامه‌هایی که به کد مدیریت شده تبدیل می‌شوند از امنیت کد پایین تری نسبت به برنامه‌های محتوی کد زبان ماشین دارند. البته برنامه‌های تحت وب و یا وب سرویس‌ها و به طور کلی برنامه‌هایی که در یک سرور مرکزی اجرا می‌شوند کمتر از این لحاظ در خطر هستند، زیرا فقط تعداد محدودی از افراد به برنامه‌های موجود در سرور دسترسی دارند و بنابراین کدها کاملاً ایمن خواهند ماند.

اما برای برنامه‌هایی که بایستی به صورت عمومی توزیع شوند، می‌توان از نرم افزارهای شرکتهای دیگر که کدهای **IL** را گنگ و نامفهوم می‌کنند استفاده کرد. این برنامه‌ها تمامی‌نامهای به کار رفته در کد (مانند نام توابع، متغیرها و ...) را تغییر می‌دهند و آنها را با نامهای بدون معنی جایگزین می‌کنند. به این ترتیب هدف و وظیفه هر تابع از روی نام آن قابل تشخیص نخواهد بود و فرد نمی‌تواند با مشاهده‌ی کد **IL** یک برنامه به سادگی از الگوریتم آن مطلع شود. شایان ذکر است که قدرت این برنامه‌ها در نامفهوم کردن کد **IL** تولید شده محدود است، زیرا اگر کد بیش از حد تغییر کند **CLR** توانایی پردازش آن را نخواهد داشت.

اگر اهمیت کد به حدی بالا است که نتوان به برنامه‌های نامفهوم کننده اطمینان کرد، می‌توان کدهای مهم برنامه را در یک کلاس مجزا قرار داد و آن را به صورت کد زبان ماشین کامپایل کرد. سپس از ویژگی **CLR** برای ارتباط کدهای مدیریت شده و کدهای مدیریت نشده برای دسترسی به مابقی کد استفاده کرد.

باید در نظر داشت که فقط زبان **IL** تمام امکانات موجود در **CLR** را ارائه می‌دهد. تمام زبانهای دیگر قسمتی از امکانات ارائه شده توسط **CLR** را شامل می‌شوند. بنابراین هنگام انتخاب زبان برنامه نویسی بایستی این نکته در نظر گرفته شود و اگر نیاز به امکاناتی از **CLR** به وجود آمد که در زبان مورد

---

[214] **IL Disassembler**

استفاده وجود نداشت می‌توان آن قسمت از برنامه را به صورت کد **IL** نوشت و یا از زبانهای دیگر که آن ویژگی را ارائه می‌دهند استفاده کرد.

نکته مهم دیگر این است که زبان **IL** به هیچ پردازنده‌ای وابسته نیست. این امر بدین معنا است که کدهای مدیریت شده در هر پلت فرمی که **CLR** برای آن ارائه شده باشد اجرا می‌شود. البته نسخه اولیه‌ی **CLR** برای ویندوزهای ۳۲بیتی عرضه شده است، اما در هر صورت این موضوع برنامه نویسان را از توجه به پردازنده‌ای که توسط کاربر مورد استفاده قرار گرفته است آزاد می‌کند.

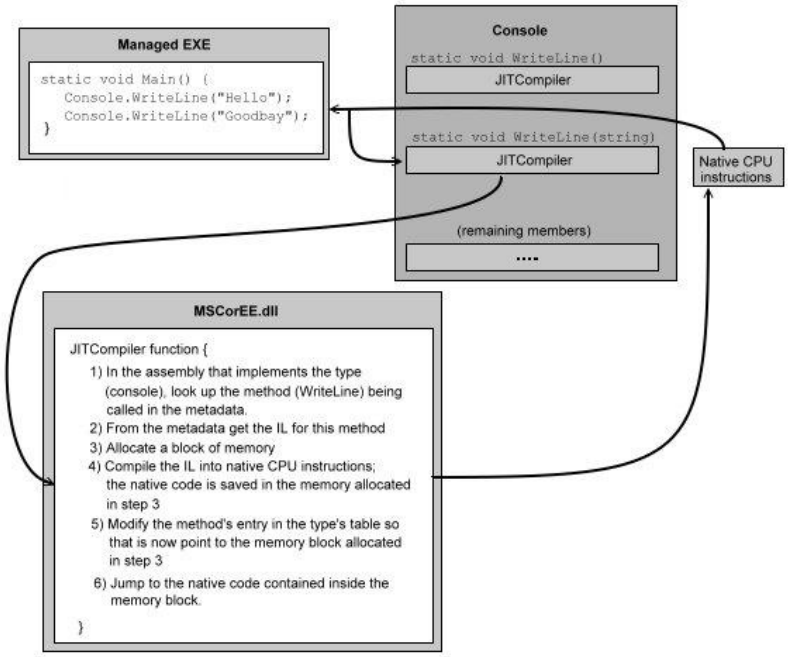
برای اجرای کدهای مدیریت شده که به زبان **IL** در فایل ذخیره شده‌اند ابتدا باید آنها را به کدهای زبان ماشین تبدیل کرد. این امر وظیفه‌ی بخشی از **CLR** به نام **JIT**[۲۱۵] است. شکل زیر مراحلی که هنگام اجرای یک تابع برای اولین بار رخ می‌دهد را نشان می‌دهد.

درست قبل از اینکه تابع **Main** اجرا شود، **CLR** نوع تمام اشیایی که توسط این تابع استفاده شده‌اند را تشخیص می‌دهد. این امر باعث می‌شود که **CLR** یک جدول داده‌ای داخلی برای مدیریت دسترسی به نوع‌های ارجاعی تشکیل دهد. برای مثال در شکل بالا تابع **Main** از کلاس ارجاعی **Console** استفاده می‌کند که این مورد توسط **CLR** در جدول داده‌ای قرار می‌گیرد. ساختار مذکور بازای هر تابع که در کلاس استفاده شده، یک ردیف اطلاعات را در جدول ایجاد می‌کند. هر ردیف شامل آدرس مکانی از حافظه است که پیاده سازی تابع در آن قرار دارد. هنگام مقدار دهی اولیه‌ی این ساختار، تمام ردیفهای آن به یکی از توابع درونی **CLR** به نام **JITCompiler** اشاره می‌کنند.

زمانی که تابع **Main** برای اولین مرتبه تابع **WriteLine** را فراخوانی می‌کند، **CLR** به جدول داده‌ای که ایجاد شده بود نگاه می‌کند تا تشخیص دهد تابعی که باید احضار کند در کدام قسمت از حافظه قرار دارد. اما همانطور که گفته شد هنگام ایجاد جدول، مقدار اولیه‌ی هر تابع به جای اینکه به آدرس خود تابع اشاره کند به آدرس تابعی به نام **JITCompiler** اشاره می‌کند. بنابراین در این قسمت به جای تابع **WriteLine**، تابع **JITCompiler** به وسیله‌ی **CLR** احضار می‌شود. این تابع وظیفه تبدیل کد **IL** به کد زبان ماشین را بر عهده دارد. به دلیل این که کدهای **IL** فقط زمانی که به آنها نیاز است به کد زبان ماشین تبدیل می‌شوند، از این قسمت از **CLR** عموماً به عنوان **JITter** و یا **JIT Compiler** یاد می‌شود.

---

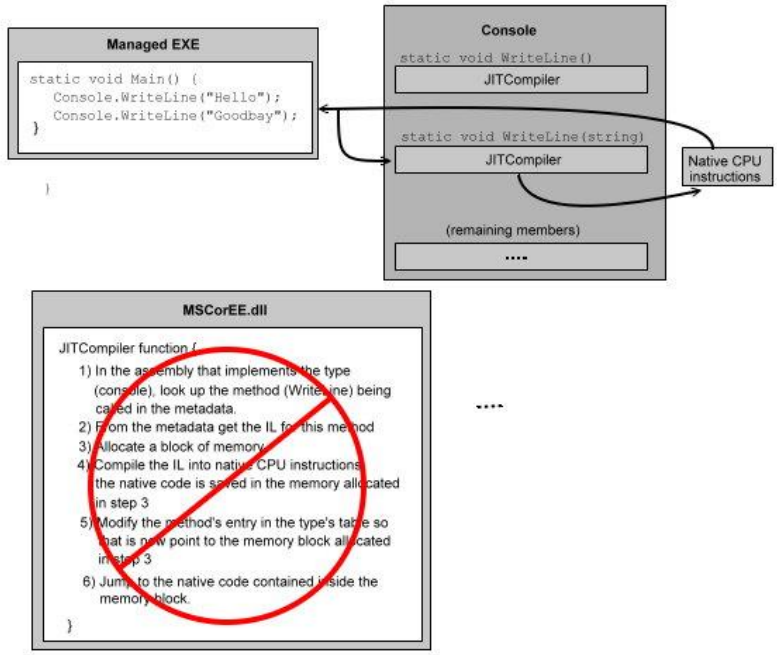
[215] **Just-In-Time Compiler**



شکل (۴-۲۵) فراخوانی تابع برای اولین بار

هنگام احضار تابع **JITCompiler**، این تابع می‌داند که چه تابعی در اصل فراخوانی شده است و این تابع در چه کلاسی قرار دارد. پس **JITCompiler** کد **IL** مربوط به تابع فراخوانی شده را بدست آورده و آن را به کد زبان ماشین تبدیل می‌کند. کد تولید شده در این مرحله در یک بلاک از حافظه که به صورت دینامیک تعیین شده است قرار می‌گیرد. در مرحله بعد، **JITCompiler** با تغییر ردیف مرتبط با تابع مذکور در جدول داده‌ای داخلی که توسط **CLR** ایجاد شده بود، آدرس موجود در ردیف را با آدرس کد زبان ماشین تولید شده جایگزین می‌کند. در نهایت **JITCompiler** به آدرس موجود در حافظه که محتوی کد تولید شده است رفته و کد را اجرا می‌کند. در پایان این تابع اجرای برنامه به تابع **Main** باز گشته و اجرای برنامه ادامه پیدا می‌کند.

با بازگشت اجرای برنامه به متد **Main**، همانطور که در شکل مشخص است تابع **WriteLine** برای دومین مرتبه فراخوانی می‌شود. در این مرتبه، کد **IL** مربوط به تابع احضار شده قبلا توسط **JITCompiler** به کد زبان ماشین تبدیل شده و در حافظه قرار گرفته است. بنابراین اجرای برنامه در این قسمت به بلاک حافظه‌ای می‌رود که کد مربوط به تابع قرار دارد و **JITCompiler** اجرا نمی‌شود.



شکل (۵-۲۵) فراخوانی تابع برای مرتبه دوم

این مورد در مرتبه اول فراخوانی یک تابع باعث ایجاد یک مکث کوتاه در برنامه می‌شود اما در مرتبه‌های بعدی احضار تابع، تابع با سرعتی برابر کدهای زبان ماشین اجرا می‌شوند. در حقیقت این کامپایل فقط یک بار در طول برنامه رخ می‌دهد.

همانطور که ذکر شد کد زبان ماشین تابع فراخوانی شده در یک بلاک حافظه دینامیک قرار میگیرد. بنابراین هنگام خاتمه برنامه کد مورد نظر از حافظه پاک می‌شود. در مراتب بعدی که برنامه اجرا می‌شود هنگام فراخوانی تابع مورد نظر، **JITCompiler** مجدداً احضار شده و تابع را به زبان ماشین تبدیل می‌کند.

طراحانی با پیش زمینه از محیطهایی مانند **C++** ممکن است عقیده داشته باشند که این عمل تاثیر منفی بر کارایی برنامه خواهد داشت. با وجود اینکه کدهای مدیریت نشده برای یک ساختار پردازنده خاص تولید می‌شوند، اما هنگام اجرا این کدها به راحتی اجرا می‌شوند. اما اجرای کدهای مدیریت شده شامل دو مرحله می‌شود. در مرحله اول سورس برنامه بایستی به کد **IL** تبدیل شود و سپس در مرحله بعد در هر بار اجرای برنامه، کد بایستی به زبان ماشین تبدیل شده و در حافظه ذخیره شود. این عمل موجب صرف بیشتر حافظه و زمان پردازنده می‌شود.

در نگاه اول ممکن است نظر این افراد درست جلوه کند اما حقیقت این است که این اعمال تاثیر چندانی بر کارایی برنامه ندارد و بررسی‌های انجام شده در این زمینه نشان میدهد که تاثیری که انجام امور ذکر شده برای اجرای کدهای مدیریت شده بر سرعت برنامه میگذارد اصلا مشهود و قابل توجه نیست.

حتی در بعضی از شرایط اجرای کدهای مدیریت شده سریعتر از کدهای مدیریت نشده انجام می‌شود. این امر به این دلیل است که کامپایلر **JIT** هنگام کامپایل برنامه‌ها نسبت به کامپایلرهای مدیریت نشده اطلاعات بیشتری از محیط اجرا در دست دارد. به طور مثال در شرایط زیر کامپایلر **JIT** سریعتر از دیگر کامپایلرها عمل می‌کند:

- **یک کامپایلر JIT از نوع ساختار پردازنده که برنامه بر روی آن اجرا می‌شود اطلاع دارد و می‌تواند از دستورات ویژه‌ای که آن نوع پردازنده ارائه می‌دهد حداکثر استفاده را داشته باشد. برای مثال این کامپایلر می‌تواند برنامه را برای پردازنده‌های Pentium4 بهینه کند. اما معمولا برنامه‌های مدیریت نشده به صورتی کامپایل می‌شوند که با تمام انواع پردازنده‌ها سازگار باشند. این امر از بهره‌وری از امکانات ویژه‌ی پردازنده جلوگیری می‌کند.**
- **یک کامپایلر JIT می‌تواند شرایطی را که همواره نادرست هستند تشخیص دهد و آنها را کامپایل نکند. برای مثال یک دستور مشابه زیر در یک تابع هرگز کامپایل نمی‌شود و این امر موجب می‌شود که کد مورد نظر برای محیطی که باید در آن اجرا شود بهینه شود**

```
If numberOfCPUs > 1 Then

        .
        .
        .
End If
```

اینها تنها تعداد محدودی از دلایلی بودند که نشان دهنده اجرای بهتر کدهای مدیریت شده نسبت به کدهای مدیریت نشده هستند. همانطور که ذکر شد سرعت این کدها برای برنامه‌ها کاملاً مناسب است. اما با وجود این اگر همچنان این احساس وجود دارد که این کدها سرعت مطلوب را ارائه نمی‌دهند می‌توان از ابزارهایی که برای تبدیل کد **IL** به کد زبان ماشین در **.NET Framework SDK** وجود دارد استفاده کرد. برای مثال ابزاری به نام **ngen.exe** وجود دارد که کد زبان ماشین مربوط به یک فایل محتوی کد **IL** را در دیسک ذخیره می‌کند. هنگامی که **CLR** مشاهده کند که کد زبان ماشین فایلی که قصد اجرای آن را دارد در دیسک موجود است از آن کد به جای کد **IL** استفاده می‌کند.

۵-۲۵ مجموعه کتابخانه کلاس NET Framework.

علاوه بر **CLR**، در مجموعه **NET Framework**. بخش دیگری نیز وجود دارد که شامل یک کتابخانه‌ی کلاس می‌شود. این بخش که **FCL**[۲۱۶] نام دارد شامل چندین هزار کلاس است که هر کدام وظیفه خاصی را انجام می‌دهند. این مجموعه با هم، یعنی **CLR** و **FCL**، به طراحان اجازه می‌دهند که چندین مدل برنامه را طراحی کنند که عبارت اند از:

- **وب سرویسهای مبتنی بر XML: وب سرویس‌ها توابعی هستند که به راحتی از طریق شبکه وب قابل دسترسی و فراخوانی هستند. این قسمت در حقیقت اصلی ترین فلسفه‌ی ظهور و ابداع NET. محسوب می‌شود.**
- **برنامه‌های تحت وب: این برنامه‌ها، برنامه‌ها و یا وب سایت‌هایی مبتنی بر صفحات HTML هستند. عموماً این برنامه‌ها با استفاده از درخواست اطلاعات از سرورهای بانک اطلاعاتی و چندین وب سرویس، اطلاعات مورد نیاز را دریافت کرده و بعد از تحلیل و انجام پردازش روی آنها، صفحات HTML مبنی بر درخواست کاربر را تشکیل داده و اطلاعات خواسته شده را از طریق مرورگر کامپیوترهای سرویس گیرنده نمایش می‌دهند. این قسمت در حقیقت لایه میانی یا لایه منطق تجاری در برنامه‌های چند لایه محسوب می‌شود.**
- **برنامه‌های تحت ویندوز: برنامه‌هایی با رابط گرافیکی برای سیستم عامل ویندوز. همانند برنامه‌های تحت وب، این برنامه‌ها نیز قدرت برقراری ارتباط با سرورهای بانک اطلاعاتی را داشته و می‌تواند از وب سرویسهای مبتنی بر XML استفاده کند. بنابراین در مواقعی که نیاز به برنامه‌های تحت وب نباشد می‌توان از این نوع برنامه‌ها با امکانات شبکه و نیز قدرت بیشتر در طراحی رابط گرافیکی استفاده کرد.**
- **برنامه‌های تحت کنسول در ویندوز: در مواقعی که نیازی به رابط گرافیکی کاربر نیست و یا یک رابط ساده کافی است می‌توان از این نوع برنامه‌ها استفاده کرد. کامپایلرها و بعضی از برنامه‌های کاربردی و نیز ابزارها از این دسته هستند.**
- **سرویسهای ویندوز: یکی دیگر از انواع برنامه‌هایی که قابلیت طراحی آن با NET. وجود دارد، سرویسهای ویندوزی است که به وسیله مرکز کنترل سرویس ویندوز (SCM)[۲۱۷] و نیز NET. Framework قابل کنترل هستند.**
- **کامپوننت‌ها و کتابخانه‌های کلاس: به وسیله NET Framework. می‌توان به طراحی کامپوننت‌هایی پرداخت که به راحتی قابل توزیع و نیز قابل استفاده در محیطهای دیگر باشد.**

---

[216] Framework Class Library
[217] Windows Service Control Manager

به دلیل اینکه **FCL** شامل چندین هزار کلاس می‌شود، تمام کلاسهای مرتبط به یکدیگر در یک فضای نام گردآوری شده‌اند. اصلی ترین فضای نام، فضای نام **System** است که محتوی کلاس **Object** و تعدادی کلاس پایه‌ای دیگر است. کلاس **Object** یک کلاس پایه است که تمام کلاسهای **FCL** از این کلاس مشتق می‌شوند. لازم به ذکر است که تمام قواعد شیئ گرایی در کلاسهای **FCL** رعایت شده‌اند و به این ترتیب این امکان را به برنامه نویس داده‌اند که در هر مرحله بتواند امکانات و یا توابع برنامه را مطابق نیازهای خود تغییر دهد و کلاسهای جدیدی طراحی کند که علاوه بر امکانات کلاسهای قبلی، نیازهای طراح و برنامه نویس را نیز به طور کامل برطرف کند.

جدول زیر بعضی از فضای نام‌های پر کاربرد در **FCL** را معرفی کرده و به طور مختصر راجع به آنها توضیح میدهد.

فضای نام	شرح
System	محتوی تمام کلاسهای پایه‌ای است که به وسیله تمام برنامه‌ها استفاده می‌شوند.
System.Collections	مجموعه‌ای است از کلاسها که برای نگهداری آرایه‌هایی از اشیا که شامل آرایه‌های عمومی مثل صف، پشته، لیست پیوندی و ... می‌شود به کار می‌رود.
System.Diagnostics	محتوی کلاس‌هایی برای مستند سازی و نیز خطا یابی در برنامه است.
System.Drawing	شامل کلاس‌هایی برای نگهداری اشیای مربوط به گرافیک دو بعدی است. این مجموعه بیشتر برای برنامه‌های تحت ویندوز و نیز نمایش عکس در برنامه‌های تحت وب به کار می‌رود.
System.EnterpriseServices	این فضای نام شامل کلاس‌هایی برای مدیریت تراکنش‌ها، فعال سازی **JIT**، امنیت و ... است که موجب کارایی بیشتر کدهای مدیریت شده در سرور می‌شوند.
System.Globalization	این قسمت شامل کلاس‌هایی برای پشتیبانی زبانهای دیگر از قبیل حروف آن زبانها، نحوه‌ی نمایش تاریخ و قالب بندی در آن و ... می‌شود.
System.IO	کلاس‌هایی برای انجام عملیات ورودی و خروجی از قبیل کار با فایلها و دایرکتوری‌ها را دربر دارد.
System.Management	شامل کلاس‌هایی برای مدیریت دیگر کامپیوترها به وسیله دستگاه‌های مدیریتی ویندوز یا **WMI** است.
System.Net	شامل کلاس‌هایی برای برقراری ارتباطات شبکه‌ای است.

System.Reflection	شامل کلاس‌هایی برای بررسی و استفاده از اطلاعات داخل **metadata** است.
System.Resources	کلاس‌هایی برای مدیریت منابع خارجی استفاده شده در برنامه را شامل می‌شود.
System.Runtime.InteropServices	کلاس‌های این فضای نام به کدهای مدیریت شده این امکان را می‌دهد که از امکانات قبلی ویندوز از قبیل کامپوننت‌های **COM** و یا توابع موجود در فایلهای **Win32** استفاده کنند.
System.Runtime.Remoting	کلاس‌هایی را شامل می‌شود که به دیگر کلاسها اجازه میدهند از را دور مورد استفاده قرار گیرند.
System.Runtime.Serialization	کلاس‌هایی را در بر دارد که به اشیا این امکان را می‌دهند به رشته‌ای تبدیل شوند که معرف آنها باشد و از رشته‌ای که معرف آنها است به یک شیئ تبدیل شوند.
System.Security	کلاس‌هایی برای محافظت از اطلاعات و منابع برنامه.
System.Text	کلاس‌هایی برای کار با متن در قالبهای مختلف از قبیل **ASCII** و یا **Unicode**.
System.Threading	شامل کلاس‌هایی است که برای اجرای همزمان پردازش‌ها و نیز هماهنگی در دسترسی به اطلاعات مورد استفاده قرار می‌گیرد.
System.Xml	کلاس‌هایی برای بررسی و پردازش داده‌ها در قالب **XML** را دربر دارد.

۶-۲۵ سیستم نوع داده‌ای عمومی

احتمالاً تا کنون متوجه این نکته شده‌اید که یکی از مهمترین قسمتهای **CLR** درباره‌ی کار با کلاسها است. این کلاسها کارایی لازم را به برنامه و یا کامپوننت در حال طراحی ارائه می‌دهند. کلاسها موجب می‌شوند که کدهای طراحی شده در یک زبان در زبانهای دیگر قابل استفاده باشد. به علت اینکه هسته اصلی **CLR** را کلاسها تشکیل می‌دهند، مایکروسافت یک سری خصوصیات عمومی در مورد کلاسها در **.NET** را تحت عنوان **CTS**[۲۱۸] عنوان کرد که شامل چگونگی تعریف کلاسها و چگونگی رفتار آنها می‌شود.

طبق خصوصیات **CTS** هر کلاس می‌تواند شامل چند عضو باشد و یا شامل هیچ عضوی نباشد. اعضای قابل تعریف در کلاس عبارت اند از:

[218] **Common Type System**

- **فیلدها:**

یک متغیر داده‌ای است که معرف وضعیت کنونی شیئ است. فیلدها به وسیله نام و نوع داده‌ای آنها مشخص میشوند.

- **متدها:**

تابعی است که پردازش خاصی را بر روی شیئ انجام می‌دهد. این پردازش معمولا شامل تغییر وضعیت آن می‌شود. متدها معمولا شامل یک نام، لیست مقادیر ورودی، لیست مقادیر خروجی و نیز عبارت‌هایی برای تعیین سطح دسترسی به آن هستند.

- **مشخصات[۲۱۹]:**

برای کاربر کلاس این عضو همانند یک فیلد است اما برای طراح کلاس این عضو همانند یک (یا دو) متد است. این نوع عضوهای کلاس به طراح و برنامه نویس این امکان را می‌دهند که قبل از قرار دادن ورودی کاربر در متغیر مربوطه، به بررسی صحت آن بپردازد و یا پردازش‌های مورد نیاز دیگر را قبل از قرار دادن مقدار انجام دهد. علاوه بر این، این نوع عضوها به طراحان اجازه ایجاد فیلدهای فقط خواندنی یا فقط نوشتنی را می‌دهند.

- **رویدادها:**

رویدادها مکانیسمی‌برای اطلاع دیگر اشیا از یک رخداد خاص در شیئ محسوب می‌شوند. برای مثال یک دکمه می‌تواند هنگامی که فشرده شد اشیای دیگر را از این اتفاق مطلع کند.

CTS علاوه بر این موارد امکاناتی را برای تعیین سطح دسترسی به اعضا ارائه می‌دهد. برای مثال اشیایی که به عنوان **public** در نظر گرفته می‌شوند توسط کلاسهای دیگر، چه در داخل و چه در خارج اسمبلی قابل دسترسی هستند. از سوی دیگر تعیین یک عضو داده‌ای به عنوان **Assembly** (در ویژوال بیسیک **Friend** نامیده می‌شود) باعث می‌شود آن عضو فقط در کلاسهای موجود در داخل اسمبلی قابل دسترس باشد.

لیست زیر سطح دسترسی‌های مختلف را برای اعضای داده‌ای کلاس تعریف می‌کند:

- **Private :**

متد مورد نظر فقط به وسیله دیگر متدهای همان کلاس قابل دسترسی است.

- **Family :**

متد مورد نظر فقط به وسیله کلاس‌هایی که از آن کلاس مشتق شده‌اند، بدون توجه به اینکه در همان اسمبلی قرار دارد یا نه، قابل استفاده است.

---

[219] **Properties**

- **Family and Assembly :**

متد مورد نظر فقط در توابعی از همان اسمبلی که علاوه بر آن از کلاس مورد نظر نیز مشتق شده‌اند قابل استفاده است. بسیاری از زبانهای برنامه نویسی از قبیل **C#** و **Visual Basic** این نوع سطح دسترسی را ارائه نمی‌دهند.

- **Assembly :**

متد مورد نظر فقط به وسیله کلاسهای داخل همان اسمبلی قابل استفاده است.

- **Family or Assembly :**

متد مورد نظر به وسیله تمام کلاس‌هایی که از این کلاس مشتق می‌شوند، در همه اسمبلی‌ها، قابل استفاده است. علاوه بر این، این متد در همه کلاسهای آن اسمبلی نیز قابل دسترسی است.

- **Public :**

متد مورد نظر د همه کلاسها صرف نظر از اینکه در چه اسمبلی قرار دارد، قابل دسترسی است.

علاوه بر این، **CTS** قواعدی را برای وراثت، طول عمر اشیا، توابع مجازی و غیره نیز بیان می‌کند.

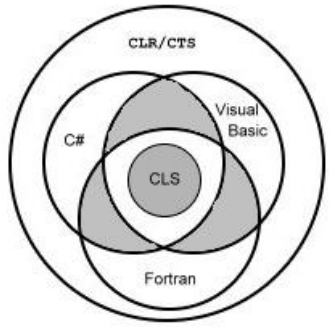
۷-۲۵ خصوصیات عمومی زبانهای برنامه نویسی

یکی از امکاناتی که همراه با **COM** عرضه شد، استفاده از کدهای نوشته شده در یک زبان به وسیله زبانی دیگر بود. این امکان به وسیله‌ی **CLR** نیز به نحوی بسیار کامل تر ارائه شده است. از سوی دیگر، **CLR** با یکی کردن تمام زبانها این امکان را می‌دهد که اشیای تولید شده در یک زبان در زبانهای دیگر قابل استفاده باشد. این قابلیت با توجه به نوع داده‌ای عمومی مشترک بین همه‌ی زبانها، استفاده از **metadata**ها برای شرح اطلاعات کلاس و محیط اجرای عمومی به وجود آمده است.

با توجه به اینکه زبانهای برنامه نویسی امکانات کاملا متفاوتی را عرضه می‌کنند، این قابلیت بسیار شگفت انگیز به نظر می‌رسد. برای مثال بعضی از زبانهای برنامه نویسی حساس به بزرگی و کوچکی حروف هستند، بعضی سربار گذاری عملگرها را پشتیبانی می‌کنند و یا بعضی این امکان را می‌دهند تا متدهای با تعداد پارامترهای نامحدود ایجاد کنیم.

اگر در ایجاد یک برنامه از امکانات خاصی یک زبان استفاده کنیم، استفاده از آن برنامه در برنامه‌های دیگر بسیار مشکل خواهد بود. بنابراین برای طراحی کلاس‌هایی که به راحتی در تمام زبانهای دیگر

قابل استفاده باشند بایستی از امکاناتی در برنامه استفاده کرد که تضمین بشود در همه زبانهای برنامه نویسی دیگر وجود دارند. برای مشخص تر کردن این امکانات که در بین همه‌ی زبانهای برنامه نویسی مشترک است، مایکروسافت یک مجموعه از خصوصیات عمومی را معین کرده که **CLS**[۲۲۰] نام دارد.
مجموعه‌ی **CLR** و **CTS**، امکاناتی بسیار بیشتر از آنچه **CLS** تعیین کرده است را ارائه می‌دهند. بنابراین اگر کلاس تحت طراحی نباید در زبانهای دیگر استفاده شود می‌توان از تمام امکانات **CTS** و **CLR** استفاده کرد. شکل زیر سعی مفهوم ذکر شده را بیان می‌کند.



شکل (۶-۲۵) تمام زبانها زیر مجموعه‌ای از تواناییهای **CLR** و تمام امکانات **CLS** را ارایه می‌دهند

همانطور که در شکل مشخص است هیچ زبانی تمامی‌امکانات **CLR** را ارائه نمی‌کند. بنابراین برای استفاده از تمام این امکانات بایستی از زبان **IL** کمک گرفت. به جز زبان **IL**، دیگر زبانها از قبیل **C#**، **Visual Basic**، **Fortran** و غیره فقط قسمتی از امکانات **CLR** و **CTS** را ارایه می‌دهند.
اگر کلاسی به منظور استفاده در دیگر زبانها طراحی می‌شود، بایستی این نکته را مد نظر قرار داد که در آن کلاس نباید از امکاناتی که خارج از محدوده **CLS** هستند استفاده کرد. در غیر این صورت آن قسمت از برنامه در زبانهای دیگر که ویژگی مورد استفاده را پشتیبانی نمی‌کنند قابل استفاده نخواهد بود.

[220] **Common Language Specifications**

ضمیمه چهار

مدیریت حافظه در .NET

در بخش قبلی، **CLR** که به عنوان هسته اصلی **.NET Framework** در نظر گرفته می‌شود، بررسی شد. به عنوان دو ویژگی مهم **CLR**، می‌توان از توانایی برقراری ارتباط آن بین زبانهای مختلف برنامه نویسی و نیز مدیریت حافظه قوی آن یاد کرد. در توضیح ویژگی اول، همانطور که در قسمت قبلی ذکر شد، **CLR** با استفاده از یک سری خصوصیات عمومی که به وسیله تمام زبانها پشتیبانی می‌شود(**CLS**)، امکان استفاده از کلاس نوشته شده در یک زبان در زبانهای دیگر را فراهم می‌کند.
ویژگی مهم دیگر **CLR** که موجب افزایش کارایی برنامه در آن می‌شود، مدیریت حافظه آن است. این بخش در **CLR** بر عهده **Garbage Collector** است که به علت اهمیت این موضوع در این قسمت کاملاً توضیح داده می‌شود.

۱-۲۶ درک مبانی کار Garbage Collector

هر برنامه به نحوی از منابع مشخصی استفاده می‌کند. این منابع می‌توانند فایلها، بافرهای حافظه، فضاهای صفحه نمایش، ارتباطات شبکه‌ای، منابع بانک اطلاعاتی و مانند اینها باشند. در حقیقت در یک محیط شیئ گرا هر نوع داده تعریف شده در برنامه بعضی از منابع موجود در برنامه را اشغال می‌کند. برای استفاده از این منابع لازم است که مقداری حافظه تخصیص داده شود. موارد زیر برای دسترسی به یک منبع مورد نیاز است:

- **تخصیص حافظه برای منبعی که لازم است مورد استفاده قرار گیرد. این تخصیص حافظه با استفاده از دستور newobj در زبان IL[221] صورت می‌گیرد که این دستور نیز، از ترجمه دستور new در زبان‌هایی مثل #C و Visual Basic و دیگر زبان‌های برنامه نویسی ایجاد می‌شود.**
- **مقدار دهی اولیه حافظه برای تنظیم حالت آغازین[۲۲۲] و قابل استفاده کردن آن. توابع Constructor در کلاسها مسئول این تنظیمات برای ایجاد حالت آغازین هستند.**
- **استفاده از منبع ایجاد شده به روش مورد نظر، برای مثال استفاده از توابع کلاس FileStream برای دسترسی به یک فایل.**

---

[221] Intermediate Language
[222] Initial State

- **از بین بردن منبع اشغال شده. در زبانهایی مانند C++ این مورد به وسیله توابع destructor در کلاس انجام می‌شد.**
- **آزاد سازی حافظه. در پلتفرم NET.، Garbage Collector مسئول کل این مرحله به شمار می‌رود.**

این مدل استفاده از منابع گرچه بسیار ساده بنظر می‌رسد، اما یکی از مهمترین عوامل خطاهای زمان اجرا در برنامه‌ها به شمار می‌رود. مواقع زیادی پیش می‌آید که برنامه نویس فراموش می‌کند بعد از استفاده از یک منبع خاص، حافظه اشغال شده به وسیله‌ی آن را آزاد کند و این حافظه تا پایان برنامه بلا استفاده باقی می‌ماند. یا مواقع زیادی پیش می‌آید که برنامه نویس در یک قسمت از برنامه حافظه مربوط به یک منبع خاص را آزاد کرده و سپس در قسمت دیگری سعی می‌کند به آن منبع دسترسی پیدا کند.

این دو باگ در برنامه‌ها از بدترین نوع خطاها به شمار می‌روند زیرا معمولاً برنامه نویس نمی‌تواند ترتیب یا زمان به وجود آمدن این خطا را پیش بینی کند. برای دیگر باگ‌ها شما می‌توانید با مشاهده رفتار اشتباه یک برنامه آن را به سادگی تصحیح کنید. اما این دو باگ موجب نشت منابع[۲۲۳] (مصرف بی جای حافظه) و از بین رفتن پایداری اشیا می‌شوند که کارایی برنامه را در زمانهای مختلف تغییر می‌دهند. برای کمک به یک برنامه نویس برای تشخیص این نوع خطاها ابزارهای ویژه‌ای مانند **Windows Task Manager** و **System Monitor ActiveX Control** و **NuMega Bounds Checker** و ... طراحی شده‌اند. اما استفاده از هر یک از این ابزارها نیز مشکلات خاص خود را دارد و نمی‌توان به نتایج بدست آمده از آنها به صورت کامل اعتماد کرد.

در اغلب موارد یک مدیریت منبع مناسب در برنامه، امری بسیار مشکل و خسته کننده است. این مورد تمرکز برنامه نویس را بر روی مطلب اصلی از بین می‌برد. به همین دلیل نیاز به یک مکانیسم که مدیریت حافظه را به بهترین نحو انجام دهد در این زمینه به وضوح احساس می‌شد. در پلت فرم **NET.** سعی شده است با استفاده از سیستمی‌به نام **Garbage Collector** تا حد زیادی از وظیفه برنامه نویس در این قسمت کاسته شود.

سیستم **Garbage Collection** سعی داشته است وظیفه کنترل استفاده از حافظه و بررسی زمان آزادسازی منابع اشغال شده در آن را کاملاً از دوش برنامه نویس بردارد. البته **Garbage Collector** درمورد منابع مورد استفاده توسط نوع داده در حافظه و نحوه آزاد سازی آنها هیچ چیز نمی‌داند. به عبارت دیگر**Garbage Collector** نمی‌داند چه طور می‌تواند مرحله ۴ از موارد بالا را انجام دهد و برنامه نویس باید توابع مربوط به این قسمت را پیاده سازی کند.

---

[۲۲۳] **Resource leaks**

باید توجه داشت که در **.NET**، فقط تعداد معدودی کلاس نیاز به نوع خاصی از آزاد سازی دارند که باید توسط برنامه نویس مشخص شود. اغلب کلاسها و یا ساختارهایی که در **.NET**، وجود دارند، از قبیل **Int32**، **Point**، **String**، **SerializationInfo** و ... به سادگی و فقط با آزاد سازی حافظه اصلی که توسط آنها اشغال شده است از بین می‌روند. به عنوان مثال، برای از بین بردن یک شئی از ساختار **Point** کافی است حافظه‌ای که به وسیله دو فیلد **X** و **Y** اشغال شده است آزاد شود.

اغلب کلاسهایی که از منابع مدیریت نشده استفاده می‌کند، مانند منابع لازم برای استفاده از یک فایل، منابع لازم برای استفاده از یک ارتباط بانک اطلاعاتی، یک سوکت، یک **Bitmap**، یک آیکن و مانند اینها همیشه به اجرای مقداری کد ویژه برای آزاد کردن حافظه اشغال شده نیاز دارند.

در پلت فرم **.NET**، برای اینکه **CLR**[224] بتواند با استفاده از سیستم **GC** حافظه یک برنامه را مدیریت کند، نیاز دارد که حافظه لازم برای تمام منابع از یک **heap** مخصوص به نام **managed heap** تخصیص داده شود. این **heap** شبیه **heap** زمان اجرای **C** است و فقط از یک لحاظ متفاوت است و آن این است که در این **heap** شما هیچ وقت حافظه تخصیص داده شده را آزاد نمی‌کنید. در حقیقت اشیا موجود در این **heap** وقتی دیگر نیازی به آنها نباشد خود به خود از بین می‌روند.

ممکن است این سوال در این قسمت ایجاد شود که چگونه **managed heap** متوجه می‌شود دیگر نیازی به یک شیئ خاص نیست؟ این مورد به وسیله‌ی الگوریتم‌های مورد استفاده در **GC** تشخیص داده می‌شود. چندین الگوریتم از **Garbage Collector** در حال حاضر در مرحله آزمایش هستند و هر کدام از این الگوریتم‌ها برای یک محیط خاص و نیز برای کسب بهترین راندمان بهینه سازی شده‌اند. در این مقاله روی الگوریتم **Garbage Collector** استفاده شده در **Microsoft .NET Framework CLR** تمکز خواهیم کرد.

زمانی که یک پروسه مقدار دهی اولیه[۲۲۵] می‌شود، **CLR** یک قسمت پیوسته از آدرس حافظه را برای آن اختصاص می‌دهد. این آدرس، فضای حافظه **managed heap** نامیده می‌شود. در این **heap** یک اشاره گر مخصوص هم وجود دارد که ما از این به بعد آن را **NextObjPtr** می‌نامیم. این اشاره گر مکان قرار گیری شیئ بعدی را در **heap** مشخص می‌کند. در ابتدا این اشاره گر به آدرس ابتدای فضای گرفته شده برای **managed heap** اشاره می‌کند.

گفتیم که دستور **newobj** در زبان **IL** باعث ایجاد یک شیئ جدید می‌شود. در بیشتر زبانها از جمله **C#** و **Visual Basic** این دستور از ترجمه دستور **new** به وجود می‌آید. این دستور **IL** باعث می‌شود که **CLR** مراحل زیر را به ترتیب انجام دهد:

---

[224] **Common Language Runtime**
[225] **Initialize**

- ☑ *محاسبه تعداد بایت‌های مورد نیاز برای کلاس یا ساختاری که باید نمونه سازی شود.*
- ☑ *اضافه کردن بایت‌های مورد نیاز برای **overhead** شیئ. هر شیئ دو فیلد **overhead** دارد: یک اشاره گر به جدول تابع و یک **SyncBlockIndex** در سیستمهای ۳۲بیتی، هر کدام از این فیلدها ۳۲ بیت هستند، که ۸ بایت را به هر شیئ اضافه می‌کند. در سیستم‌های ۶۴ بیتی، هر کدام از این فیلدها ۶۴ بیت است که ۱۶ بایت را برای هر شیئ اضافه می‌کند.*
- ☑ *سپس **CLR** چک می‌کند که حافظه مورد نیاز برای شیئ جدید در **managed heap** موجود باشد. اگر فضای کافی موجود باشد این شیئ در آدرسی که **NextObjPtr** به آن اشاره می‌کند ایجاد شده، تابع **constructor** شیئ مذکور فراخوانی می‌شود (اشاره گر **NextObjPtr** به عنوان پارامتر **this** به **constructor** فرستاده می‌شود) و دستور **newobj** آدرس شیئ ایجاد شده را برمی‌گرداند. درست قبل از اینکه آدرس برگردانده شود، **NextObjPtr** به بعد از شیئ ایجاد شده پیشروی می‌کند و مثل قبل آدرسی که باید شیئ بعدی در آن قرار گیرد را در خود ذخیره می‌کند.*

شکل زیر یک **managed heap** که سه شیئ **A** و **B** و **C** در آن ذخیره شده است را نشان می‌دهد. اگر یک شیئ جدید ایجاد شود این شیئ دقیقا در جایی که **NextObjPtr** به آن اشاره می‌کند قرار می‌گیرد (درست بعد از شیئ **C**).

Newly intialized managed heap with three objects constructed in it

شکل (۱-۲۶) **Managed heap**

همانطور که مشاهده کردید این **heap** با **heap** زمان اجرای **C** عملکرد متفاوتی دارد و به روش دیگری مدیریت می‌شود. در یک **heap** زمان اجرای **C**، تخصیص حافظه برای یک شی به حرکت در میان ساختارهای داده از یک لیست پیوندی نیاز دارد. زمانی که یک بلاک حافظه با اندازه لازم پیدا شد این بلاک حافظه تقسیم شده و شیئ مذکور در آن ایجاد می‌شود و اشاره گرهای موجود در لیست پیوندی برای نگه داری در آن شیئ نیز تغییر داده می‌شوند. برای **managed heap** تخصیص حافظه برای یک شیئ به معنای اضافه کردن یک مقدار به اشاره گر است. در حقیقت تخصیص حافظه به یک شیئ در **managed heap** تقریباً به سرعت ایجاد یک متغیر در **stack** است! به علاوه در بیشتر **heap**ها مانند **heap** زمان اجرای **C** حافظه در جایی اختصاص داده می‌شود که فضای خالی کافی یافت شود. بنابراین اگر چند شیئ بلافاصله بعد از هم در برنامه ایجاد شوند، ممکن است این اشیا چندین مگا بایت آدرس حافظه با هم فاصله داشته باشند ولی در **managed heap** ایجاد چند شیئ بلافاصله بعد از هم باعث قرار گرفتن ترتیبی این اشیا در حافظه می‌شود.

ایجاد اشیا به صورت متوالی در حافظه اهمیت زیادی دارد. در بیشتر برنامه‌ها وقتی برای یک شیئ حافظه در نظر گرفته می‌شود که یا بخواهد با یک شیئ دیگر ارتباط داشته باشد یا بخواهد چندین بار در یک قطعه کد استفاده شود. برای مثال وقتی یک حافظه برای شیئ **BinaryWriter** ایجاد شد بلافاصله بعد از آن یک حافظه برای شیئ **FileStream** گرفته می‌شود. سپس هنگامی که برنامه از **BinaryWriter** استفاده می‌کند در حقیقت به صورت درونی از شیئ **FileStream** هم استفاده می‌کند. در یک محیط کنترل شده به وسیله **Garbage Collector** برای اشیای جدید به صورت متوالی فضا در نظر گرفته می‌شود که این عمل موجب افزایش راندمان به دلیل موقعیت ارجاع‌ها می‌شود. این مورد به این معنی است که مجموعه کارهای پروسه کمتر شده و این نیز متشابه قرار گرفتن اشیای مورد استفاده توسط برنامه در **CPU Cache** است.

تا کنون اینگونه به نظر می‌رسید که **managed heap** در عمل بسیار بهتر از **heap** زمان اجرای **C** است و این نیز به دلیل سادگی پیاده سازی و سرعت آن است. اما نکته دیگری که اینجا باید در نظر گرفته شود این است که **managed heap** این توانایی‌ها را به این دلیل به دست می‌آورد که یک فرض بزرگ انجام می‌دهد و آن فرض این است که فضای آدرس و حافظه بینهایت هستند. به وضوح این فرض کمی‌خنده دار به نظر می‌رسد و مسلماً **managed heap** باید مکانیسم ویژه‌ای را به کار برد تا بتواند این فرض را انجام دهد. این مکانیسم **Garbage Collector** نامیده می‌شود ، که در ادامه درباره‌ی نحوه کارکرد آن صحبت خواهیم کرد.

زمانی که یک برنامه عملگر **new** را فراخوانی می‌کند ممکن است فضای خالی کافی برای شیئ مورد نظر وجود نداشته باشد. **heap** این موضوع را با اضافه کردن حجم مورد نیاز به آدرس موجود در **NextObjPtr** متوجه می‌شود. اگر نتیجه از فضای در نظر گرفته شده برای برنامه تجاوز کرد **heap** پر شده است و **Garbage Collector** باید آغاز به کار کند.

نکته: *مطالبی که ذکر شد در حقیقت صورت ساده شده مسئله بود. در واقعیت یک* ***Garbage Collection*** *زمانی رخ می‌دهد که نسل صفر کامل شود. بعضی* ***Garbage Collector*** *ها از نسل‌ها استفاده می‌کنند که یک مکانیسم به شمار می‌رود و هدف اصلی آن افزایش کارایی است. ایده اصلی به این صورت است که اشیای تازه ایجاد شده نسل صفر به شمار می‌روند و اشیایی قدیمی‌تر در طول عمر برنامه در نسل‌های بالاتر قرار می‌گیرند. جداسازی اشیا و دسته بندی آنها به نسل‌های مختلف می‌تواند به* ***Garbage Collector*** *اجازه دهد اشیایی موجود در نسل خاصی را به جای تمام اشیا مورد بررسی قرار دهد. در بخش‌های بعدی نسل‌ها با جزئیات تمام شرح داده می‌شوند. اما تا ان مرحله فرض می‌شود که* ***Garbage collector*** *وقتی رخ می‌دهد که* ***heap*** *پر شود.*

۲-۲۶ الگوریتم Garbage Collection

همانطور که ذکر شد، زمانی که **heap** مورد استفاده در برنامه پر شود سیستم **GC** فراخوانی می‌شود.**Garbage Collection** نیز بررسی می‌کند که آیا در **heap** شیئ وجود دارد که دیگر توسط برنامه استفاده نشود. اگر چنین اشیای در برنامه موجود باشند حافظه گرفته شده توسط این اشیا آزاد می‌شود (اگر هیچ حافظه‌ای برای اشیای جدید در **heap** موجود نباشد خطای **OutOfMemoryException** توسط عملگر **new** رخ میدهد). اما چگونه **Garbage Collector** تشخیص می‌دهد که آیا برنامه یک متغیر را نیاز دارد یا خیر؟ همانطور که ممکن است تصور کنید این سوال پاسخ ساده‌ای ندارد.

هر برنامه دارای یک مجموعه از **root**ها است. یک **root** اشاره گری است به یک نوع داده ارجاعی. این اشاره گر یا به یک نوع داده ارجاعی در **managed heap** اشاره می‌کند یا با مقدار **Nothing** مقدار دهی شده است. همچنین تمام متغیرهای استاتیک و یا عمومی[۲۲۶] یک **root** به شمار می‌روند . به علاوه هر متغیر محلی که از نوع ارجاع باشد و یا پارامترهای توابع در **stack** نیز یک **root** به شمار می‌روند. در نهایت، درون یک تابع، یک ثبات **CPU** که به یک شیئ از نوع ارجاع اشاره کند نیز یک **root** محسوب می‌شود.

زمانی که کامپایلر **JIT** یک کد **IL** را کامپایل می‌کند علاوه بر تولید کدهای **Native** یک جدول داخلی نیز تشکیل می‌دهد. منطقاً هر ردیف از این جدول یک محدوده از بایت‌های آفست را در دستورات محلی **CPU** برای تابع نشان می‌دهند و برای هر کدام از این محدوده‌ها یک مجموعه از آدرس‌های حافظه یا ثبات‌های **CPU** را که محتوی **root**ها هستند مشخص می‌کند. برای مثال جدول ممکن است مانند جدول زیر باشد:

Sample of a JIT compiler-produced table showing mapping of native code offsets to a method's roots

Starting Byte Offset	Ending Byte Offset	Roots
0x00000000	0x00000020	this, arg1, arg2, ECX, EDX
0x00000021	0x00000122	this, arg2, fs, EBX
0x00000123	0x00000145	Fs

اگر زمانی که کدی بین آفست **0x00000021** و **0x00000122** در حال اجرا است **heap** پر شود، **Garbage Collector** شروع به کار می‌کند. در این مرحله **Garbage Collector** می‌داند که

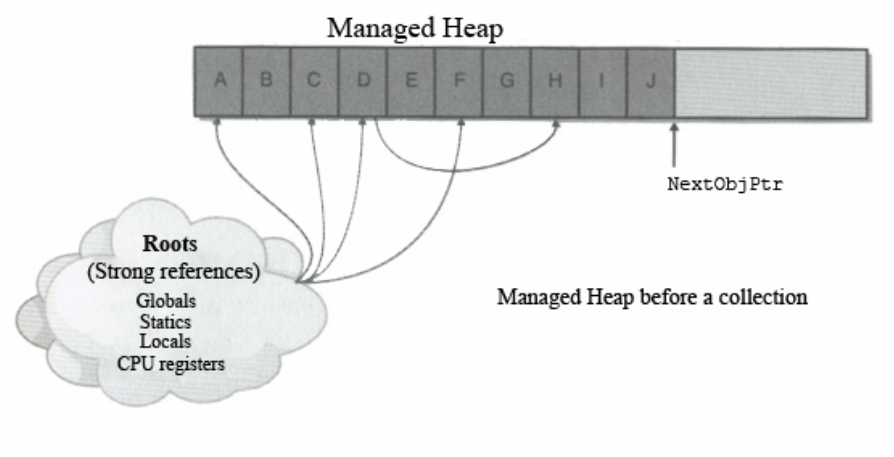
[226] Global Variables

پارامترهای **this**و **arg2** و متغیرهای محلی **fs** و ثبات **EBX** همه **root** هستند و به اشیایی درون **heap** اشاره می‌کنند که نباید به عنوان اشیای زائد تلقی شوند. به علاوه **Garbage Collector** می‌تواند بین **stack** حرکت کند و با امتحان کردن جدول داخلی هر کدام از این توابع، **root**ها را برای تمام توابع فراخوانی شده مشخص کند.

نکته: *در جدول بالا توجه کنید که آرگومان **arg1** در تابع بعد از دستورات **CPU** در آفست **0x00000020** دیگر به چیزی اشاره نمی‌کند و این امر بدین معنی است که شیئی که **arg1** به آن اشاره می‌کند هر زمان بعد از اجرای این دستورات می‌تواند توسط **Garbage Collector** جمع آوری شود (البته فرض بر اینکه هیچ شیئ دیگری در برنامه به شیئ مورد ارجاع توسط **arg1** اشاره نمی‌کند). به عبارت دیگر به محض اینکه یک شیئ غیر قابل دسترسی باشد برای جمع آوری شدن توسط **Garbage Collector** داوطلب می‌شود و به همین علت باقی ماندن اشیا تا پایان یک متد توسط **Garbage Collector** تضمین نمی‌شود.*

با وجود این، زمانی که یک برنامه در حالت **debug** اجرا شده باشد و یا ویژگی **DebuggableAttribute** به اسمبلی برنامه اضافه شده باشد و یا اینکه پارامتر **isJITOptimizeDisabled** با مقدار **true** در **constructor** برنامه تنظیم شده باشد، کامپایلر **JIT** طول عمر تمام متغیرها را، چه از نوع ارجاعی و چه از نوع مقدار، تا پایان محدوده شان افزایش می‌دهد که معمولاً همان پایان تابع است. این افزایش طول عمر از جمع آوری شدن متغیرها توسط **Garbage Collector** در محدوده اجرایی آنها در طول برنامه جلوگیری می‌کند و این عمل فقط در زمان **debug** یک برنامه مفید واقع می‌شود.

زمانی که **Garbage Collector** شروع به کار می‌کند، فرض می‌کند که تمام اشیای موجود در **heap** زائد هستند. به عبارت دیگر فرض می‌کند که هیچ کدام از **root**های برنامه به هیچ شیئ در **heap** اشاره نمی‌کنند. سپس **Garbage Collector** شروع به حرکت در میان **root**های برنامه می‌کند و یک گراف از تمام **root**های قابل دسترسی تشکیل می‌دهد. برای مثال **Garbage Collector** ممکن است یک متغیر عمومی را که به یک شیئ در **heap** اشاره می‌کند پیدا کند. شکل زیر یک **heap** را با چندین شیئ تخصیص داده شده نشان میدهد. همانطور که در شکل مشخص است **root**های برنامه فقط به اشیای **A** و **C** و **D** و **F** به طور مستقیم اشاره می‌کنند. بنابراین تمام این اشیا از اعضای گراف محسوب می‌شوند. زمان اضافه کردن شیئ **Garbage Collector.D** متوجه می‌شود که این شیئ به شیئ **H** اشاره می‌کند، بنابراین شیئ **H** نیز به گراف برنامه اضافه می‌شود و به همین ترتیب **Garbage Collector** تمام اشیای قابل دسترسی در **heap** را مشخص می‌کند.



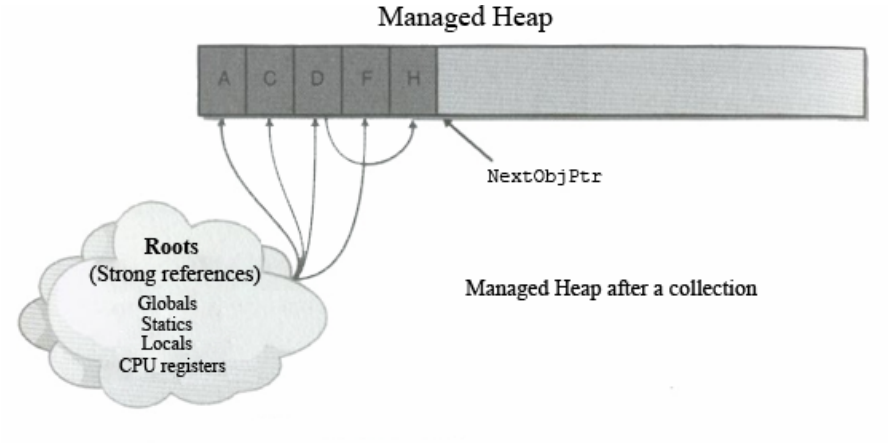
شکل (۲-۲۶)

زمانی که این بخش از گراف کامل شد، **Garbage Collector** به بررسی **root**های بعدی می‌پردازد و مجدداً اشیا را بررسی می‌کند. در طول اینکه **Garbage Collector** از یک شیئ به یک شیئ دیگر منتقل می‌شود، اگر سعی کند که یک شیئ تکراری را به گراف اضافه کند، حرکت در آن مسیر را متوقف می‌کند. این نوع رفتار دو هدف را دنبال می‌کند: اول اینکه **Garbage Collector** از هیچ شیئ دو بار عبور نمی‌کند بنابراین راندمان برنامه به شکل قابل توجهی افزایش پیدا می‌کند. دوم اینکه هر قدر هم در برنامه لیست‌های پیوندی دایره‌ای از اشیا موجود باشند ، **Garbage Collector** در حلقه‌های بینهایت نمی‌ماند.

زمانی که تمام **root**ها بررسی شدند، گراف **Garbage Collector** محتوی تمام اشیایی است که به نحوی از طریق **root**های برنامه قابل دسترسی می‌باشند و هر شیئ که در گراف نباشد به این معنی است که توسط برنامه قابل دسترسی نیست و یک شیئ زائد محسوب می‌شود. بعد از این **Garbage Collector** به صورت خطی **heap** را طی می‌کند و دنبال بلاک‌های پیوسته از اشیای زائد مشخص شده توسط گراف می‌گردد (که هم اکنون فضای خالی محسوب می‌شوند). اگر بلاک‌های کوچکی پیدا شوند **Garbage Collector** این بلاک‌ها را به همان حال قبلی رها می‌کند.

اگر یک بلاک پیوسته وسیع یافت شد، **Garbage Collector** اشیای غیر زائد را به سمت پایین حافظه **heap** شیفت داده (این کار با تابع استاندارد **memcopy** انجام می‌شود) و به این طریق **heap** را فشرده می‌کند. طبیعتاً حرکت دادن اشیا به سمت پایین در **heap** باعث نا معتبر شدن تمام اشاره گرهای موجود به آن اشیا می‌شود. حال **Garbage Collector** مسئول است که مقدار تمام اشاره گرها را مجدداً تنظیم کند. بعد از این که این عمل فشرده سازی روی **heap** انجام شد **NextObjPtr** به آخرین

شیئ غیر زائد اشاره می‌کند. شکل زیر یک **managed heap** را بعد از اتمام کار **Garbage Collector** نشان می‌دهد.



شکل (۳-۲۶)

همانطور که در شکل می‌بینید،**Garbage Collector** افزایش بازدهی قابل توجهی را ایجاد می‌کند. اما به یاد داشته باشید زمانی **Garbage Collector** شروع به کار می‌کند که نسل صفر کامل شود و تا آن زمان **managed heap** به صورت قابل توجهی سریعتر از **heap** زمان اجرای **C** است. به علاوه الگوریتم **Garbage Collector**که در **CLR** استفاده می‌شود به صورتی بهینه سازی شده است که راندمان کاری را به مقدار زیادی افزایش می‌دهد.

کد زیر نشان می‌دهد که چگونه به اشیا حافظه تخصیص داده شده و مدیریت می‌شوند:

```
Module Module1
        Sub Main()
                ' ArrayList object created in heap, a is now a root
                Dim a As New ArrayList()

                ' Create 10000 objects in the heap

                For index As Integer = 1 To 10

                Next

                For x As Int32 = 0 To 10000
                        a.Add(New Object())      ' Object created in heap
                Next

                ' Right now, a is a root (on the thread's stack). So a is
```

```
            ' reachable and the 10000 objects it refers to
            ' are reachable.
            Console.WriteLine(a.Count)

            ' After a.Length returns, a isn't referred to in the code
            ' andis no longer a root. If another thread were to start
            ' a garbage collection before the result of a.Length were
            ' passed to WriteLine, the 10001 objects would have their
            ' memory reclaimed.

            Console.WriteLine("End Of Method")
        End Sub
End Module
```

نکته: *اگر فکر می‌کنید که **Garbage Collector** یک تکنولوژی با ارزش محسوب می‌شود، ممکن است تعجب کنید که چرا در **ANSI C++** قرار نمی‌گیرد. دلیل این مورد این است که **Garbage Collector** احتیاج دارد که **root**های موجود در برنامه را تعیین هویت کند و بداند که هر کدام دقیقاً به چه نوع شیئ اشاره می‌کنند. مشکل با **C++** مدیریت نشده این است که این زبان تغییر نوع یک اشاره گر را از یک نوع به یک نوع دیگر مجاز می‌داند و هیچ راهی برای فهمیدن این که این اشاره گر به چه چیز اشاره می‌کند وجود ندارد. در **CLR**، **managed heap** همیشه می‌داند که نوع واقعی یک شیئ چیست و از اطلاعات **metadata** برای مشخص کردن اینکه کدام عضوها از یک شی به اشیای دیگر اشاره می‌کنند استفاده می‌کند.*

۳-۲۶ ارجاع‌های ضعیف

زمانی که یک **root** به یک شیئ اشاره می‌کند شیئ نمی‌تواند توسط **Garbage Collector** جمع آوری شود چون ممکن است برنامه به آن دسترسی پیدا کند. زمانی که یک **root** به یک شیئ اشاره می‌کند، اصطلاحاً گفته می‌شود که یک ارجاع قوی به آن شیئ موجود است. در کنار ارجاعات قوی، **Garbage Collector** از نوع دیگری از ارجاعات به نام ارجاعات ضعیف هم پشتیبانی می‌کند. ارجاعات ضعیف به **Garbage Collector** اجازه می‌دهند که شیئ را جمع آوری کند و همچنین به برنامه اجازه دهد که به آن شیئ دسترسی داشته باشد.

اگر فقط ارجاعات ضعیف به یک شیئ موجود باشند و **Garbage Collector** آغاز به کار کند آن شیئ از **heap** پاک می‌شود و سپس زمانی که برنامه سعی کند به آن دسترسی پیدا کند این عمل با شکست مواجه می‌شود. به عبارت دیگر برای دسترسی به یک ارجاع ضعیف برنامه باید یک ارجاع قوی به آن داشته باشد. اگر برنامه قبل از اینکه **Garbage Collector** آن را جمع آوری کند یک ارجاع قوی به آن

شیئ تشکیل دهد، **Garbage Collector** نمی‌تواند این شیئ را حذف کند چون یک ارجاع قوی به آن شیئ موجود است.

اجازه دهید معنی واقعی موضوع را در کد بررسی کنیم:

```
Sub SomeMethod()
        ' Create a string reference to a new Object.
        Dim o As New Object()

        ' Create a strong reference to a short WeakReference object.
        ' The WeakReference object tracks the Object's lifetime.
        Dim wr As New WeakReference(o)

        o = Nothing        ' Remove the strong reference to the object.

        o = wr.Target
        If o = Nothing Then
                ' A garbage collection occurred and Object's memory was
                ' reclaimed.
        Else
                ' A garbage collection didn't occur and I can successfully
                ' access the Object using o.
        End If
End Sub
```

ممکن است این سوال ایجاد شود که استفاده از ارجاعات ضعیف چه مزیتی را ایجاد می‌کند؟ خوب بعضی از ساختار داده‌ها به سادگی ایجاد می‌شوند اما مقدار زیادی حافظه را اشغال می‌کنند. برای مثال ممکن است شما برنامه‌ای داشته باشید که به لیستی از تمام دایرکتوری‌ها و فایلهای موجود در کامپیوتر نیاز داشته باشد. شما به سادگی می‌توانید یک درخت از تمام این اطلاعات تشکیل دهید و به محض اینکه برنامه شما اجرا شد به جای مراجعه به‌هارد به این درخت در حافظه مراجعه کند این امر سرعت برنامه شما را افزایش می‌دهد.

اما مشکل این است که این ساختار داده حجم بسیار زیادی از حافظه را اشغال می‌کند. اگر کاربر بخش دیگری از برنامه را آغاز کند، این درخت ممکن است دیگر مورد نیاز نباشد اما مقدار زیادی از حافظه را اشغال کرده است. شما می‌توانید ارجاع اصلی به این درخت در حافظه را رها کنید. اما اگر کاربر به قسمت اول برنامه برگشت، مجدداً نیاز به بازسازی این درخت خواهید داشت. ارجاعات ضعیف این امکان را می‌دهد که به بهترین نحو این موقعیت را کنترل کنید.

زمانی که کاربر از قسمت اول برنامه خارج شود، می‌توانید یک ارجاع ضعیف به **root** مربوط به این درخت در حافظه ایجاد کنید و تمام ارجاعات قوی به آن را از بین ببرید. اگر حافظه برای بخشهای دیگر برنامه کم شد، **Garbage Collector** حافظه گرفته شده توسط درخت مذکور را می‌گیرد و زمانی

که کاربر مجدداً به قسمت اول برنامه برگشت و برنامه سعی کرد از ارجاع ضعیف یک ارجاع قوی بدست آورد، این تلاش شکست می‌خورد و برنامه مجدداً درخت را تشکیل می‌دهد. در غیر این صورت برنامه از همان درخت قبلی موجود در حافظه استفاده می‌کند.

کلاس **System.WeakReference** دو **Constructor** عمومی را ارائه می‌دهد:

public New(ByVal target As Object)
public New(ByVal target As Object, ByVal trackResurrection As Boolean)

پارامتر **target** شیئ که **WeakReference** باید نگهداری کند را مشخص می‌کند. پارامتر **trackResurrection** مشخص می‌کند که آیا **WeakReference** باید شیئ را حتی بعد از **finalize** شدن هم نگه داری کند یا خیر که معمولاً مقدار این پارامتر **false** است.

برای راحتی، یک ارجاع ضعیف که بعد از **finalize** شیئ هم می‌توان آن را بازیابی کرد ارجاع ضعیف بلند و ارجاع ضعیفی را که بعد از **finalize** شدن نمی‌توان آن را بازیابی کرد ارجاع ضعیف کوتاه می‌نامیم. اگر نوع داده شیئ تابع **finalize** را ارئه ندهد ارجاع ضعیف کوتاه و بلند مانند هم عمل می‌کنند. اما به شدت توصیه می‌شود که از به کار بردن ارجاعات ضعیف خودداری کنید، زیرا این نوع ارجاعات به شما اجازه می‌دهند که حتی بعد از **finalize** شدن هم به یک شئی دسترسی داشته باشید که می‌تواند موجب به وجود آمدن یک موقعیت غیر قابل پیش بینی در برنامه شود.

یک بار که یک ارجاع ضعیف را برای یک شیئ ایجاد کردید عموماً باید تمام ارجاعات قوی به آن را برابر **Nothing** قرار دهید، در غیر این صورت **Garbage Collector** توانایی جمع آوری این شیئ را در مواقع ضروری نخواهد داشت.

برای استفاده مجدد شیئ باید ارجاع ضعیف را به یک ارجاع قوی تبدیل کرد. برای این کار می‌توان از **WeakReference.Target** استفاده کرد و شیئ را به یکی از **root**های برنامه اختصاص داد. اگر این عبارت مقدار **Nothing** را برگرداند، یعنی شیئ مذکور توسط **Garbage Collector** جمع آوری شده است، در غیر این صورت **root** حاوی آدرس یک ارجاع قوی به این مقدار محسوب می‌شود و برنامه از این طریق می‌تواند شیئ مذکور را کنترل کند.

۴-۲۶ نسلها[۲۲۷]

همانطور که پیشتر ذکر شد نسلها مکانیسمی‌درون **CLR Garbage Collector** به شمار می‌روند که هدف اصلی آن بهبود کارایی برنامه است. یک **Garbage Collector** که با مکانیسم نسلها کار می‌کند

[227] **Generations**

(همچنین به عنوان **Garbage Collector** زودگذر[۲۲۸] هم نامیده می‌شود) فرضهای زیر را برای کار خود در نظر می‌گیرد:

- **هر چه یک شیئ جدیدتر ایجاد شده باشد طول عمر کوتاهتری هم خواهد داشت.**
- **هر چه یک شیئ قدیمیتر باشد طول عمر بلندتری هم خواهد داشت.**
- **جمع آوری قسمتی از heap سریعتر از جمع آوری کل آن است.**

مطالعات زیادی معتبر بودن این فرضیات را برای مجموعه بزرگی از برنامه‌های موجود تایید کرده‌اند و این فرضیات بر طرز پیاده سازی **Garbage Collector** تاثیر داشته‌اند. در این قسمت طرز کار این مکانیسم شرح داده شده است.

زمانی که یک **managed heap** برای بار اول ایجاد می‌شود دارای هیچ شیئ نیست. اشیایی که به **heap** اضافه شوند در نسل صفر قرار می‌گیرند. اشیای موجود در نسل صفر اشیای تازه ایجاد شده‌ای هستند که تا کنون توسط **Garbage Collector** بررسی نشده‌اند. تصویر زیر یک برنامه را که تازه آغاز به کار کرده است نشان میدهد که دارای پنج شیئ است(از **A** تا **E**). بعد از مدتی اشیای **C** و **E** غیر قابل دسترسی می‌شوند.

A Newly initialized heap containing some objects; all in generation 0. No .collections have occured now

شکل (۴-۲۶)

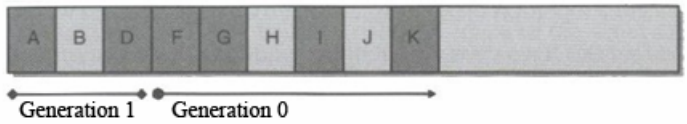
زمانی که **CLR** آغاز به کار می‌کند یک مقدار نهایی را برای نسل صفر در نظر می‌گیرد که به طور پیش فرض ۲۵۶ کیلو بایت است(البته این مقدار ممکن است تغییر کند). بنابراین اگر شیئ بخواهد ایجاد شود و در نسل صفر فضای کافی وجود نداشته باشد **Garbage Collector** آغز به کار می‌کند. اجازه دهید تصور کنیم که اشیای **A** تا **E** ۲۵۶ کیلو بایت فضا اشغال کرده‌اند زمانی که شیئ **F** بخواهد تشکیل شود **Garbage Collector** باید آغاز به کار کند. در این هنگام**Garbage Collector** تشخیص می‌دهد که اشیای**C** و **E** زائد محسوب می‌شوند و شیئ **D** باید فشرده شود. بنابراین این شیئ به کنار شیئ **B** میرود. اشیایی که بعد از این مرحله باقی میمانند (اشیای **A** و **B** و **D**) وارد نسل یک میشوند. اشیای موجود در نسل یک به این معنی هستند که یک بار توسط **Garbage Collector** بررسی شده‌اند. **Heap** برنامه مفروض بعد از اولین مرحله به صورت زیر در می‌آید.

[228] **Ephemeral Garbage Collector**

After one collection; generation 0 survivors are promoted to generation 1
Generation 0 is empty

شکل (۵-۲۶)

بعد از یک بار اجرای **Garbage Collector** هیچ شیئ در نسل صفر باقی نمی‌ماند. مثل همیشه اشیایی که بعد از این ایجاد می‌شوند به نسل صفر اضافه می‌شوند. شکل زیر اجرای برنامه و به وجود آمدن اشیای **F** تا **K** را نشان می‌دهد. به علاوه در طول اجرای برنامه اشیای**B** و **H** و **J** غیر قابل استفاده شده‌اند و حافظه گرفته شده توسط آنها باید آزاد شود.



New objects are collected in generation 0

شکل (۶-۲۶)

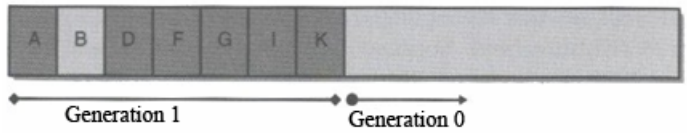
حال فرض کنیم با تخصیص حافظه برای شیئ **L** در نسل صفر مقدار داده‌های موجود در این نسل از ۲۵۶کیلوبایت فراتر رود. چون نسل صفر به سر حد خود رسیده است **Garbage Collector** باید آغاز به کار کند. زمانی که عمل **Garbage Collection** آغاز می‌شود ابتدا باید مشخص شود که کدام نسل باید مورد بررسی قرار گیرد. پیشتر ذکر شد زمانی که **CLR** آغاز به کار می‌کند برای نسل صفر ۲۵۶ کیلو بایت فضا اختصاص می‌دهد. همچنین **CLR** یک سر حد نیز برای نسل یک در نظر میگیرد. فرض می‌کنیم این مقدار فضا شامل ۲ مگا بایت باشد.

زمانی که عمل **Garbage Collection** انجام می‌شود،**Garbage Collector** بررسی می‌کند که چه مقدار فضا توسط نسل یک اشغال شده است. در این حالت نسل یک فضایی کمتر از ۲ مگا بایت را اشغال کرده است بنابراین **Garbage Collector** فقط نسل صفر را مورد بررسی قرار می‌دهد. یک بار دیگر فرضیات **Garbage Collector** را که در ابتدا ذکر شد مرور کنید. اولین فرض این بود که اشیای تازه ایجاد شده دارای عمر کوتاه تری هستند. بنابراین این گونه به نظر میرسد که نسل صفر دارای بیشترین مقدار اشیای زائد باشد و جمع آوری حافظه از این نسل موجب آزاد سازی مقدار زیادی حافظه می‌شود. بنابراین **Garbage Collector** نسل یک را رها می‌کند و فقط به جمع آوری نسل صفر می‌پردازد که این عمل موجب افزایش سرعت کارکرد پروسه **Garbage Collector** می‌شود.

به وضوح، رها سازی اشیای نسل یک موجب افزایش سرعت و کارایی **Garbage Collector** می‌شود. اگر یک **root** یا یک شیئ از این نسل به شیئ از نسل قدیمی‌تر اشاره کند (برای مثال یک **Root** از نسل صفر به یک شیئ از نسل ۱ اشاره کند)، **Garbage Collector** می‌تواند از این نوع ارجاعات صرف نظر کند، و بدین وسیله زمان مورد نیاز را برای تشکیل گرافی از اشیای قابل دسترس کاهش دهد.
البته ممکن است که یک شیئ قدیمی‌به یک شیئ جدید اشاره کند. برای اطمینان از این که این شیئ قدیمی‌نیز مورد بررسی قرار می‌گیرد **Garbage Collector** از یک مکانیسمهای داخلی **JIT** استفاده می‌کند. به این ترتیب زمانی که فیلد ارجاع یک شیئ تغییر کرد، **JIT** یک بیت خاص را تنظیم می‌کند. این پشتیبانی توسط **JIT** باعث می‌شود که **Garbage Collector** بتواند تشخیص دهد کدام از اشیای قدیمی‌از زمان آخرین عمل جمع آوری تا کنون تغییر کرده‌اند و فقط اشیای قدیمی‌که دارای فیلدهای تغییر کرده هستند را بررسی می‌کند.

نکته: *تستهای کارایی انجام شده نشان می‌دهند که عمل **Garbage Collection** در نسل صفر کمتر از یک میلی ثانیه در یک کامپیوتر پنتیوم با سرعت ۲۰۰ مگا هرتز زمان می‌برد.*

یک **Garbage Collector** که از نسلها استفاده می‌کند همچنین فرض می‌کند اشیایی که مدت زیادی است که در حافظه مانده‌اند به زودی نیز از حافظه خارج نمی‌شوند. بنابراین این احتمال میرود که اشیای موجود در نسل یک همچنان در طول برنامه قابل دسترس خواهند بود. بنابراین اگر **Garbage Collector** اشیای موجود در نسل یک را بررسی کند احتمالاً مقدار زیادی متغیر غیر قابل دسترسی در برنامه نخواهد یافت و حافظه زیادی را آزاد نخواهد کرد. بنابراین آزاد سازی نسل یک چیزی جز اتلاف وقت نخواهد بود. اگر هر شیئ زائدی در نسل یک به وجود بیاید در همان نسل باقی خواهد ماند. بعد از اجرای عملیات ذکر شده شکل **heap** به صورت زیر در می‌آیند.

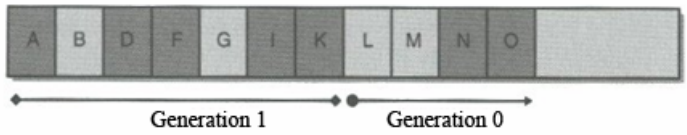


After two collections; Generation 0 survivors are promoted to Generation 1 (growing the size of Generation 1); Generation 0 is empty

شکل (۷-۲۶)

همانطور که می‌بینید تمام اشیای که از نسل صفر باقی مانده‌اند وارد نسل یک شده‌اند. چون **Garbage Collector** نسل یک را بررسی نمی‌کند شیئ **B** حافظه‌ای را که گرفته است آزاد نمی‌کند با وجود اینکه از آخرین عمل **Garbage Collector** تا کنون این متغیر در برنامه قابل استفاده نبوده است.

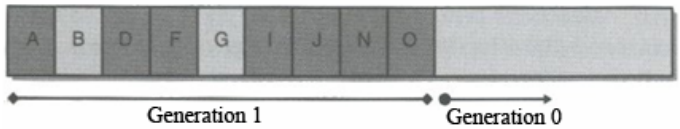
مجدداً بعد از پایان کار **GC**، نسل صفر دارای هیچ شیئ نخواهد بود و بنابراین مکانی برای قرارگیری اشیای جدید به وجود می‌آید. در ادامه، برنامه به کار خود ادامه می‌دهد و اشیای **L** تا **O** را ایجاد می‌کند و در حال اجرا برنامه استفاده از اشیای **G** و **L** و **M** را پایان داده و آنها را غیر قابل دسترس می‌کند. بنابراین **heap** به صورت زیر در می‌آید:



New objects are allocated in generation 0. Generation 1 has more garbage

شکل (۸-۲۶)

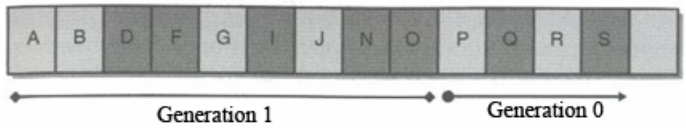
اجازه دهید فکر کنیم که تخصیص حافظه برای شیئ **P** باعث تجاوز نسل صفر از سر حد خود شود و این عمل موجب اجرای مجدد **Garbage Collector** شود. چون تمام اشیای موجود در نسل یک کمتر از ۲ مگا بایت است **Garbage Collector** مجدداً تصمیم می‌گیرد که فقط نسل صفر را بررسی کند و از اشیای غیر قابل دسترسی در نسل یک چشم پوشی کند(اشیای **B** و **G**). بعد از عمل جمع آوری **heap** به صورت زیر در می‌آید.



After three collections; Generation 0 survivors are promoted to generation 1 (growing the size of generation 1 again) Generation 0 is empty

شکل (۹-۲۶)

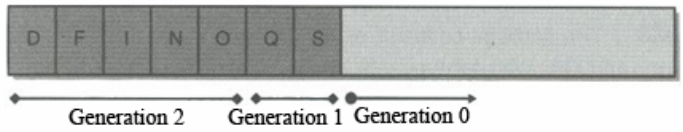
در تصویر بالا مشاهده می‌کنید که نسل یک به مرور در حال رشد و افزایش حجم است. اجازه دهید تصور کنیم که اشیای موجود در نسل یک ۲ مگا بایت فضای قابل استفاده در نسل یک را اشغال کرده‌اند. در این مرحله، برنامه مراحل اجرای خود را همچنان ادامه می‌دهد و در این مرحله اشیای **P** تا **S** تولید می‌شوند که این اشیا نسل صفر را نیز تا سر حد خود پر می‌کنند. **Heap** در این مرحله مشابه شکل زیر می‌شود.



New objects are allocated in generation 0; Generation 1 has more garbage

شکل (۱۰-۲۶)

زمانی که برنامه سعی در تخصیص حافظه برای شیئ **T** دارد نسل صفر کاملاً پر است و **Garbage Collector** باید آغاز به کار کند. این مرتبه، اشیای موجود در نسل یک هر ۲ مگا بایت فضای خود را تا سر حد فضای نسل یک اشغال کرده‌اند. علاوه بر اشیای موجود در نسل صفر، تصور می‌شود که بعضی از اشیای موجود در نسل یک هم به صورت غیر قابل استفاده در آمده‌اند. بنابراین این مرتبه، **Garbage Collector** تصمیم می‌گیرد که تمام اشیای موجود در نسل یک و نسل صفر را مورد بررسی قرار دهد. بعد از اینکه هر دو نسل به طور کامل توسط **Garbage Collector** مورد بررسی قرار گرفتند، **heap** به صورت شکل زیر در می‌آید.



After four collections; Generation1 survivors are promoted to Generation 2; Generation 0 survivors are promoted to generation 1, and generation 0 is empty

شکل (۱۱-۲۶)

مانند قبل، اشیایی که در این مرحله از جمع آوری از نسل صفر باقی ماندند وارد نسل یک می‌شوند و نیز اشیایی که در این مرحله از نسل یک باقی ماندند وارد نسل دو می‌شوند. مثل همیشه، بلافاصله نسل صفر از اشیا خالی می‌شود و اشیای جدید می‌توانند در این قسمت قرار گیرند. اشیای موجود در نسل دو اشیای هستند که حداقل دو بار توسط **Garbage Collector** مورد بررسی قرار گرفته‌اند. ممکن است بعد از یک یا دو بار جمع آوری نسل صفر، با انجام این عمل در نسل یک، مقداری حافظه آزاد شود، اما این عمل تا زمانی که نسل یک به سر حد خود نرسد انجام نمی‌شود که این کار ممکن است نیاز به چندین بار اجرای جمع آوری در نسل صفر باشد.

Managed heap فقط سه نسل را پشتیبانی می‌کند: نسل صفر، نسل یک و نسل دو. بنابراین چیزی به نام نسل سه وجود ندارد. زمانی که **CLR** آغاز به کار می‌کند، سر حدهایی را برای هر سه نسل در نظر می‌گیرد. همانطور که پیشتر ذکر شد، سر حد برای نسل صفر حدود ۲۵۶کیلوبایت است، سر حد برای نسل یک حدوداً ۲ مگا بایت است و سر حد برای نسل دو حدود ۱۰ مگا بایت است. بنابراین سر حد نسلها به گونه‌ای انتخاب شده است که موجب افزایش بازدهی و راندمان برنامه شود. هرچه سر حد یک نسل بیشتر باشد عمل **Garbage Collection** کمتر روی آن نسل صورت می‌گیرد و دوباره، باعث بهبود کارایی می‌شود که به دلیل فرضیات اولیه است: اشیای جدید دارای طول عمر کوتاهتری هستند، اشیای قدیمی‌طول عمر بیشتری دارند.

Garbage Collector موجود در **CLR** در واقع یک جمع آوری کننده با تنظیم کننده خودکار است. این بدین معنا است که **Garbage Collector** از رفتار برنامه شما می‌آموزد که چه زمانی باید عمل

جمع آوری را انجام دهد. برای مثال اگر برنامه شما اشیای زیادی را ایجاد کند و از آنها برای مدت زمان کوتاهی استفاده کند، این امر ممکن است که آزاد سازی حافظه در نسل صفر مقدار زیادی حافظه را آزاد کند.حتی ممکن است تمام حافظه گرفته شده در نسل صفر آزاد شود.

اگر **Garbage Collector** مشاهده کند که بعد از انجام جمع آوری نسل یک تعداد محدودی از اشیا باقی ماندند، ممکن است که تصمیم بگیرد که سر حد نسل صفر را از ۲۵۶ کیلو بایت به ۱۲۸ کیلو بایت کاهش دهد. این کاهش در فضای معین به این معنی است که عمل جمع آوری باید در فواصل زمانی کوتاه تری رخ دهد اما فضای کمتری را بررسی کند. بنابراین کارهای پروسه شما به صورت قابل توجهی افزایش نمی‌یابد. اگر تمام اشیای موجود در نسل صفر زائد محسوب شوند دیگر احتیاجی به فشرده سازی حافظه توسط **Garbage Collector** نیست. این عمل می‌تواند به سادگی با آوردن اشاره گر **NextObjPtr** به ابتدای حافظه مورد نظر برای نسل صفر انجام شود. این عمل به سرعت حافظه را آزاد می‌کند!

نکته:Garbage Collector *به بهترین نحو با برنامه‌های* ***ASP.NET*** *و سرویسهای وب مبتنی بر* ***XML*** *کار می‌کند. برای برنامه‌های تحت* ***ASP.NET****، یک تقاضا از طرف کلاینت می‌رسد، یک جعبه از اشیای جدید تشکیل می‌شود، اشیا کارهای تعیین شده توسط کلاینت را انجام می‌دهند، و نتیجه به سمت کلاینت بر میگردد. در این مرحله تمام اشیای موجود برای انجام تقاضای کلاینت زباله تلقی می‌شوند. به بیان دیگر، هر تقاضای برنامه‌های تحت* ***ASP.NET*** *باعث ایجاد حجم زیادی از زباله میشوند. چون این اشیا اغلب بلافاصله بعد از ایجاد دیگر قابل دسترسی نیستند هر عمل جمع آوری موجب آزاد سازی مقدار زیادی از حافظه می‌شود. این کار مجموعه کارهای پروسه را بسیار کاهش میدهد بنابراین راندمان* ***Garbage Collector*** *محسوس خواهد بود.*

به بیان دیگر، اگر **Garbage Collector** نسل صفر را مورد بررسی قرار دهد و مشاهده کند که مقدار زیادی از اشیا وارد نسل یک شدند، مقدار زیادی از حافظه توسط **Garbage Collection** آزاد نمی‌شود، بنابراین **Garbage Collector** سر حد نسل صفر را تا ۵۱۲ کیلو بایت افزایش می‌دهد. در این مرحله **GC** کمتر انجام می‌شود اما با هر بار انجام این عمل مقدار زیادی حافظه آزاد می‌شود.

در طول این قسمت چگونگی تغییر دینامیک سر حد نسل صفر شرح داده شد. اما علاوه بر سر حد نسل صفر سر حد نسلهای یک و دو نیز بر اساس همین الگوریتم تغییر می‌کنند. به این معنی که زمانی که این نسلها مورد عمل جمع آوری قرار می‌گیرند **Garbage Collector** بررسی می‌کند که چه مقدار فضا آزاد شده است و چه مقدار از اشیا به نسل بعد رفته‌اند. بر اساس نتایج این بررسیها **Garbage Collector** ممکن است ظرفیت این نسلها را کاهش یا افزایش دهد که باعث افزایش سرعت اجرای برنامه می‌شود.

۵-۲۶ دیگر عوامل کارایی Garbage Collector

پیشتر در این مقاله الگوریتم کار **Garbage Collector** شرح داده شد. با این وجود در طول این توضیحات یک فرض بزرگ صورت گرفته بود: اینکه فقط یک ترد[۲۲۹] در حال اجرا است. اما در مدل واقعی چندین ترد به **managed heap** دسترسی دارند و یا حداقل اشیای قرار گرفته در **managed heap** را تغییر می‌دهند. زمانی که یک ترد موجب اجرای عمل جمع آوری توسط **Garbage Collector** می‌شود، دیگر تردها حق دسترسی به اشیای موجود در **managed heap** را ندارند(این مورد شامل ارجاع‌های اشیای موجود در **stack** هم می‌شود) زیرا **Garbage Collector** ممکن است مکان این اشیا را تغییر دهد.

بنابراین وقتی **Garbage Collector** بخواهد عمل جمع آوری را آغاز کند، تمام تردهایی که در حال اجرای کدهای مدیریت شده[۲۳۰] هستند به حال تعلیق در می‌آیند.**CLR** دارای چندین مکانیسم نسبتاً متفاوت است که می‌تواند تردها را به حالت تعلیق در آورد بنابراین عمل جمع آوری می‌تواند به درستی اجرا شود. دلیل اینکه **CLR** از چندین مکانیسم استفاده می‌کند به حالت اجرا نگاه داشتن تردها تا حداکثر زمان ممکن و کاهش سربار کردن آنها در حافظه تا حداقل زمان ممکن است. تشریح این مکانیسم‌ها از اهداف این مقاله خارج است اما تا این حد لازم است ذکر شود که مایکروسافت فعالیتهای زیادی را برای کاهش فشار پردازشی ناشی از **Garbage Collector** انجام داده است. و نیز این مکانیسم‌ها به سرعت در حال تغییر هستند تا به بهترین کارایی خود برسند.

زمانی که **CLR** میخواهد **Garbage Collector** را اجرا کند، ابتدا تمام تردها در پروسه جاری را که در حال اجرای کدهای مدیریت شده هستند به حال تعلیق در آورده و **CLR** برای تعیین موقعیت هر ترد تمام اشاره گرهای دستورات در حال اجرا توسط تردها را بررسی می‌کند. سپس برای تعیین اینکه چه کدی توسط ترد در حال اجرا بوده آدرس اشاره گر دستور با جدول ایجاد شده توسط کامپایلر**JIT** مقایسه می‌شود.

اگر دستور درحال اجرا توسط ترد در یک آفست مشخص شده به وسیله جدول مذکور باشد گفته می‌شود که ترد به یک نقطه امن دسترسی دارد. یک نقطه امن نقطه‌ای است که در آنجا می‌توان بدون هیچ مشکلی ترد را به حال تعلیق در آورد تا **Garbage Collector** کار خود را آغاز کند.اگر اشاره گر دستور در حال اجرای ترد در روی یک آفست مشخص شده توسط جدول درونی تابع قرار نداشت، بنابراین ترد در یک نقطه امن قرار ندارد و **CLR** نمی‌تواند **Garbage Collector** را اجرا کند. در این حالت**CLR** ترد راهایجک[۲۳۱] می‌کند: به این معنی که **CLR** استک مربوط به ترد را به گونه‌ای تغییر

---

[229] **Thread**
[230] **Managed Code**
[231] **Hijack**

می‌دهد که آدرس بازگشت به یک تابع خاص پیاده سازی شده درون **CLR** اشاره کند. سپس ترد به ادامه کار خود بازمی‌گردد. زمانی که متد در حال اجرا توسط ترد ادامه پیدا کند، این تابع ویژه اجرا خواهد شد و ترد به حالت تعلق درخواهد آمد.

با وجود این ممکن است در بعضی مواقع ترد از متد خود باز نگردد. بنابراین زمانی که ترد به اجرای خود ادامه می‌دهد، **CLR** ۲۵۰میلی ثانیه صبر می‌کند. سپس دوباره بررسی می‌کند که آیا ترد به یک نقطه امن طبق جدول **JIT** رسیده است یا نه. اگر ترد به یک نقطه امن رسیده بود **CLR** ترد را به حالت تعلیق درمی‌آورد و **Garbage Collector** را اجرا می‌کند در غیر این صورت مجدداً سعی می‌کند با تغییر **Stack** مربوط به ترد اجرای آن را به تابع دیگری انتقال دهد در صورت شکست مجدداً **CLR** برای چند میلی ثانیه دیگر نیز صبر می‌کند. زمانی که تمام تردها به یک نقطه امن رسیدند یا اینکه با موفقیت‌هایجک شدند، **Garbage Collector** می‌تواند کار خود را آغاز کند. زمانی که عمل جمع آوری انجام شد تمام تردها به وضعیت قبلی خود بر میگردند و اجرای برنامه ادامه پیدا می‌کند. تردهای هایجک شده هم به متدهای اولیه خود بازمی‌گردند.

نکته: *این الگوریتم یک پیچ خوردگی کوچک دارد. اگر **CLR** یک ترد را به حالت تعویق درآورد و دریابد که ترد در حال اجرای یک کد مدیریت نشده[۲۳۲] بود آدرس بازگشت ترد هایجک می‌شود و به ترد اجازه داده می‌شود که به اجرای خود ادامه دهد. با این وجود در این حالت به **Garbage Collector** اجازه داده می‌شود که اجرا شود در حالی که ترد مذکور در حال اجرا است. این مورد هیچ اشکالی را به وجود نمی‌آورد زیرا کدهای مدیریت نشده به اشیای موجود در **managed heap** دسترسی ندارند تا زمانی که آن اشیا پین[۲۳۳] شوند. یک شیئ پین شده شی است که **Garbage Collector** حق حرکت دادن آن را در **managed heap** ندارد. اگر تردی که در حال حاضر در حال اجرای یک کد مدیریت نشده بود، شروع به اجرای یک کد مدیریت شده کند، ترد هایجک می‌شود و به حالت تعلیق درمی‌آید تا زمانی که **Garbage Collection** به درستی به اتمام برسد.*

علاوه بر مکانیسمهای ذکر شده(نسلها، نقاط امن، و هایجک کردن)، **Garbage Collector** از بعضی از مکانیسمهای اضافی دیگری نیز استفاده می‌کند که باعث افزایش بازدهی آن می‌شود.

۱-۵-۲۶ اشیای بزرگ

فقط یک نکته قابل ذکر دیگر که باعث افزایش سرعت و بازدهی بهتر می‌شود باقی مانده است. هر شیئ که ۸۵۰۰۰ بایت یا بیشتر فضای حافظه را اشغال کند یک شیئ بزرگ در نظر گرفته می‌شود. اشیای

[232] **Unmanaged Code**
[233] **Pin**

بزرگ در یک **heap** ویژه اشیای بزرگ قرار می‌گیرند. اشیای درون این **heap** مانند اشیای کوچک (که راجع به آنها صحبت شد) **finalize** و آزاد می‌شوند. با این وجود این اشیا هیچ وقت تحت فشرده سازی قرار نمی‌گیرند زیرا شیفت دادن ۸۵۰۰۰ بایت بلاک حافظه درون **heap** مقدار زیادی از زمان **CPU** را هدر می‌دهد.

اشیای بزرگ همواره به عنوان نسل دو در نظر گرفته می‌شوند، بنابراین این اشیا باید فقط برای منابعی که مدت زمان زیادی در حافظه می‌مانند ایجاد شوند. تخصیص اشیایی که دارای طول عمر کوتاه هستند در قسمت اشیای بزرگ باعث می‌شود که عمل جمع آوری نسل دو سریعتر انجام شود و این مورد نیز به بازدهی و کارایی برنامه صدمه وارد می‌کند.