**GERENCIAR FLUXO DE PROGRAMA (25–30%)**

Processamento multithreading e assíncrono

* Usar a biblioteca Task Parallel, incluindo o Parallel.For method, PLINQ, Tasks; gerar threads usando ThreadPool; desbloquear a UI; usar palavras-chave async e await; gerenciar dados usando coleções simultâneas

**ASYNCHRONOUS PROGRAMMING**

Os primeiros computadores foram criados seguindo um design lógico chamado “arquitetura von Newmann”, desenvolvido por John von Newmann e outros matemáticos em 1945. Segundo esse design, um computador deve ter uma unidade de processamento, uma unidade de controle, memória e um sistema de entrada e saída (IO). A unidade de processamento e a unidade de controle formam a unidade central de processamento (CPU). Como o design tinha apenas uma unidade de processamento, os programas que precisavam ser escritos para esse tipo de design eram seqüenciais e a maioria das linguagens de programação foi criada para ser usada de maneira sequencial, prática ainda usada nas linguagens de programação atuais, incluindo C#. A maior desvantagem de criar esses aplicativos é que, sempre que seu aplicativo precisava esperar que algo acontecesse, todo o sistema congelava, criando uma experiência do usuário muito desagradável.

Em 1967, Gordon Moore observou que o número de transistores que podem ser ajustados na mesma superfície em um chip de silício está dobrando a cada dois anos. Hoje, essa duplicação acontece a cada um ano e meio. Até 2005, isso se traduzia em várias melhorias, como dobrar a frequência e a velocidade de processamento, dobrar a capacidade, reduzir o tamanho do chip pela metade e assim por diante. Em 2005, a frequência em que uma CPU poderia operar atingiu um platô. Embora os fabricantes de hardware ainda possam seguir a lei de Moore, a frequência não pode ser aumentada sem grandes implicações, principalmente devido ao enorme calor gerado pelo processador, e esse calor precisava ser resolvido. Portanto, para se beneficiar ainda do avanço tecnológico descrito pela lei de Moore, o fabricante de hardware começou a fornecer mais unidades de processamento por processador, conhecidas como núcleos, em vez de aumentar a velocidade. Os desenvolvedores agora enfrentam o maior desafio de suas carreiras, porque muitos estão acostumados a pensar e desenvolver aplicativos para máquinas que possuem apenas um núcleo. Felizmente, a Microsoft já percebeu isso, então criou novas bibliotecas e introduziu novos paradigmas no C# para que a vida pudesse ser mais fácil. Threads foram introduzidos para minimizar esse tipo de problema

**MPROCESSING > MPROGRAMMING > MTASKING > MTHREADING**

**MULTIPROCESSAMENTO**

Multi processamento é quando um computador utiliza vários processadores para realizar uma tarefa, processo ou programa simultaneamente.Assim, a tarefa pode ser executada em paralelo. Diminui o tempo de computação. Exemplo: Ao criar um arquivo, o computador usa a hora e a data como padrão.

**MULTIPROGRAMAÇÃO**

É usado em sistemas operacionais em lote para reduzir o desperdício da CPU, por alternância de contexto de processo. Através de paralelismo lógico e de quantização de tempo (timesharing) a técnica de multiprogramação permite executar várias tarefa/programa/trabalho/processo ao mesmo tempo.

* **Processo**>> Execução de um programa, **dinâmica e efêmera** (requer mais ou menos recursos, altera estado na execução).
* **Programa**>> Entidade **estática** (Guardado na memória como arquivo comum) e permanente.

Apenas um programa por vez é capaz de obter a CPU para executar suas instruções, enquanto todos os outros estão esperando a sua vez. Assim, quando termina a sua execução (isto é, no estado de execução) e espera pelo estado periférico (isto é, estado de espera) o sistema operacional seleciona um trabalho a partir do pool de trabalho e inicia sua execução de acordo com um algoritmo de agendamento; quando esse trabalho precisa esperar por operações I/O, a CPU carrega e trabalha em outra tarefa que não está esperando por I/O e pronta para o processo.

Isso está em contraste com a multitarefa, em que cada tarefa recebe um intervalo de tempo (também chamado **Quantum**) para sua execução.Tanto a multiprogramação como a tarefa são o mesmo conceito de tarefa de comutação no processador, a diferença está no conceito e na razão da comutação.

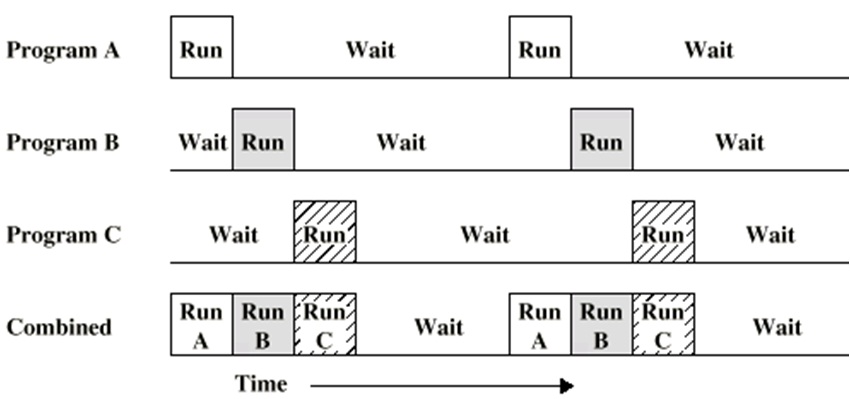
Ter um processador significava que apenas um thread poderia ser executado a qualquer momento. Isso pode ser alcançado de duas maneiras diferentes:

Todos os algoritmos classificam os processos em estados: Iniciando, Pronto, Executando, Entrada/Saída e Terminado. Ter um processador significava que apenas um thread poderia ser executado a qualquer momento. Isso pode ser alcançado de duas maneiras diferentes:

1. **Preemptivo**>> Gentil, respeita prioridade de processos. O sistema operacional possui um componente chamado agendador que garante que nenhum thread monopolize a CPU. É assim que o Windows é implementado.
2. **Não preemptivo**>> não respeito FIFO ou SJF(Shortest *Job First*). todo thread deve abrir mão do controle para que outro thread possa ser executado.

Os algoritmos comuns de escalonamento de sistemas em lote incluem:

|  |  |
| --- | --- |
| **Escalonamento** | **Descrição** |
| [FIFO](https://pt.wikipedia.org/wiki/FIFO)(First In, first Out) | Onde o primeiro que chega será o primeiro a ser executado |
| [FCFS](https://pt.wikipedia.org/w/index.php?title=FCFS&action=edit&redlink=1) (First Come, first Served) |
| [SJF](https://pt.wikipedia.org/wiki/SJF) (Shortest Job First) | Onde o menor processo ganhará a CPU. |
| [SRT](https://pt.wikipedia.org/wiki/SRT) (Shortest Remaining Time) | Neste algoritmo é escolhido o processo que possua o menor tempo restante, mesmo que esse processo chegue à metade de uma operação, se o processo novo for menor ele será executado primeiro. |
| [Algoritmo Loteria](https://pt.wikipedia.org/w/index.php?title=Algoritmo_Loteria&action=edit&redlink=1) | O Sistema Operacional distribui tokens (fichas), numerados entre os processos, para o escalonamento é sorteado um numero aleatório para que o processo ganhe a vez na CPU, processos com mais tokens têm mais chance de receber antes a CPU |
| Algoritmo de Prioridade | Cada processo no estado de pronto recebe uma prioridade, os processos com maiores prioridades são executados primeiro. |
| [Escalonamento garantido](https://pt.wikipedia.org/wiki/Escalonamento_garantido) | Se houver N usuários ativos na rede, cada um vai receber em torno de 1/N da capacidade de processamento que um usuário usou para todos os seus processos desde o momento em que tal usuário tornou-se ativo. |
| Round-Robin | Nesse escalonamento o [sistema operacional](https://pt.wikipedia.org/wiki/Sistema_operacional) possui um timer, chamado de [quantum](https://pt.wikipedia.org/wiki/Quantum), onde todos os processos ganham o mesmo valor de quantum para rodarem na [CPU](https://pt.wikipedia.org/wiki/CPU), depois que o quantum acaba e o processo não terminou, ocorre uma preempção e o processo é inserido no fim da fila. Com exceção do algoritmo RR, FIFO e garantido, todos os outros sofrem do problema de [Inanição](https://pt.wikipedia.org/wiki/Inani%C3%A7%C3%A3o_(computa%C3%A7%C3%A3o)) (starvation), preemptivo; |
| [Múltiplas Filas](https://pt.wikipedia.org/wiki/M%C3%BAltiplas_Filas) | São usadas várias filas de processos prontos para executar, cada processo é colocado em uma fila, e cada fila tem uma política de escalonamento própria. |
| Algoritmo Fair-Share | O escalonamento é feito considerando o dono dos processos, onde cada usuário recebe uma fração da CPU e processos são escalonados procurando garantir essa fração. Se um usuário A possui mais processos que um usuário B e os dois têm a mesma prioridade, os processos de A serão mais demorados que os do B. |



Exemplo : Executar um navegador(chrome, Firefox) e enviar um e-mail e executar aplicativo do sistema (MS Word, calculadora ou Excel simultaneamente.

**MULTITASKING**

Isto é basicamente multiprogramação no contexto de um ambiente interativo **single-user** ou máquina com um único processador, em que o sistema operacional alterna entre várias **tarefas (programa, processo, threads)** na memória principal. No sistema multitarefa, também chamado de compartilhamento de tempo, a CPU alterna de uma tarefa para a próxima tarefa (compartilhando recursos comuns, como CPU e memória) tão rapidamente que aparece como todas as tarefas estão sendo executadas ao mesmo tempo. Ou eja, em hardwares equipados com uma única CPU, cada thread é processada de forma aparentemente simultânea, pois a mudança entre uma thread e outra é feita de forma tão rápida que para o utilizador, isso está acontecendo paralelamente. Em hardwares com múltiplos CPUs ou multi-cores, as threads são realizadas realmente de forma simultânea e paralela.

O conceito de multitarefa é bastante semelhante a multiprogramação, mas diferença é que, no multitarefa, o usuário está realmente envolvido com trabalhos diferentes de uma só vez, ou seja, a comutação entre os trabalhos/processos ocorre com tanta freqüência que os usuários podem interagir com cada programa enquanto ele é executado.

Uma tarefa em um sistema multitarefa não é um programa aplicativo inteiro, mas pode se referir a um "thread de execução" quando um processo é dividido em subtarefas, multitarefa não implica paralelismo. Cada tarefa menor não rouba a CPU até que ela termine, eles compartilham uma pequena quantidade do tempo de CPU chamado **Quantum**. Os sistemas operacionais multiprogramação e multitarefa são sistemas de compartilhamento de tempo.

Algoritmos de escalonamento comuns usados ​​para multitarefa são: Round-Robin, Priority Scheduling (várias filas), Shortest-Process-Next.

Exemplo: ouvir música, jogar jogo, trabalhar no MS Word, Excel e outros aplicativos simultaneamente

**CRIANDO APLICAÇÕES RESPONSIVAS**

Menos de 20 anos atrás, a maioria dos sistemas operacionais de consumidor (SO) poderia executar um único processo com um thread de execução. (Um thread é a menor unidade de execução que pode ser agendada independentemente pelo sistema operacional.) Em um sistema operacional de **Monothread**, o computador executa apenas um aplicativo por vez. Normalmente, havia um intérprete de linha de comando que estava interpretando os comandos inseridos pelo usuário. Quando um comando foi inserido, o intérprete transferiu o controle para o processador para o aplicativo ao qual o comando estava se referindo. Quando o aplicativo foi concluído, ele transferiu o controle de volta para o intérprete. Se você pensar bem, isso faz muito sentido, considerando o fato de que você tinha apenas um thread. O maior problema era que o usuário podia sentir que o computador congelou quando um aplicativo fez uma das seguintes coisas:

* **Cálculos intensivos:** Quando seu aplicativo precisava fazer cálculos intensivos, não havia muito o que fazer, exceto usar um computador mais rápido para diminuir o tempo necessário para fazer o cálculo ou dividir o problema em outros menores e distribuí-lo por vários computadores, ambos com operações caras e, às vezes, pode levar mais tempo para fazer os cálculos.
* **Obter dados da E/S:** Quando seu aplicativo buscava dados da E/S, sua CPU aguardava a chegada dos dados, sem fazer nenhum processamento no enquanto isso.

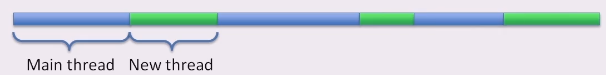
**Thread**

Thread é a unidade de execução básica de um programa em execução. Nas versões atuais do Windows, cada aplicativo é executado em seu próprio processo. Um processo isola um aplicativo de outros aplicativos, fornecendo sua própria memória virtual e garantindo que diferentes processos não possam se influenciar. Cada processo é executado em seu próprio thread. Um thread é algo como uma CPU virtualizada. Se um aplicativo trava ou atinge um loop infinito, apenas o processo do aplicativo é afetado. A Thread é possui seu próprio contador de programa, conjunto do registrador, pilha mas compartilha o código, os dados e o arquivo do processo ao qual pertence, permitindo que um processo tenha vários threads ao mesmo tempo ativos no **mesmo espaço de endereço.**

* **Monothread:** Os sistemas que suportam uma única thread (em real execução), uma thread deve aguardar a conclusão de outra ação.



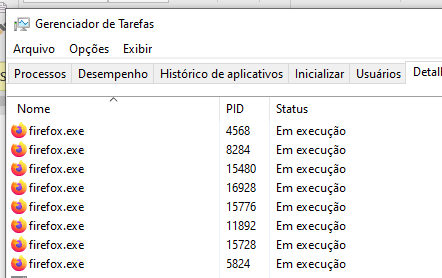
* **Multithread**: os sistemas que suportam múltiplas threads;
  + Compartilham o ambiente de execução de um único processo
  + Em uma CPU de núcleo único, o processo alterna entre threads.



* + Em uma CPU com vários núcleos, os dois threads podem executar em paralelo



Exemplo: O melhor exemplo de thread são as guias de um navegador. Se você tem 5 abas que estão sendo abertas e usadas, então o programa realmente cria 5 threads do programa (multi-threading).



O agendador do Windows funciona da seguinte maneira:

1. Cada thread recebe uma prioridade atribuída quando é criada. Um thread criado não é iniciado automaticamente; você tem que fazer isso.
2. Quando um thread é iniciado, ele será adicionado a uma fila com todos os thread que podem ser executados.
3. O planejador pega o thread com a maior prioridade na fila e começa a executá-lo.
4. Se vários thread tiverem a mesma prioridade, o planejador os agendará em ordem circular (round robin).
5. Quando o tempo alocado terminar, o planejador suspende o thread, adicionando-o no final da fila. Depois disso, ele pega um novo thread para executá-lo.
6. Se não houver outro thread com prioridade mais alta que o recém-interrompido, esse thread será executado novamente.
7. Quando um thread é bloqueado porque precisa aguardar uma operação de E/S ou por outros motivos, como bloqueio, o thread será removido da fila e outro thread será agendado para execução.
8. Quando o motivo do bloqueio termina, o thread é adicionado novamente na fila para ter a chance de ser executado.
9. Quando um thread termina o trabalho, o planejador pode escolher outro thread para executá-lo.

Há um thread chamado processo ocioso do sistema que não faz nada, exceto manter o processador ocupado quando não há outro thread para executar. Esse processo de divisão do tempo cria a impressão de que seu sistema operacional pode executar vários aplicativos ao mesmo tempo, incluindo responder aos comandos da interface do usuário que você envia, como mover o mouse ou mover as janelas, etc. A seguir, é apresentada uma lista com alguns desses threads:

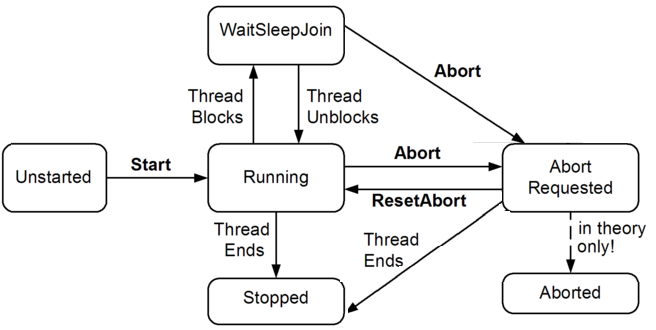
* **Garbage Collector thread**: Responsável pela coleta de lixo.
* **Finalizer thread**: responsável por executar o método Finalize de seus objetos.
* **Main thread**: responsável por executar o método do seu aplicativo principal.
* **UI thread:** se seu aplicativo for um aplicativo Windows Form, um aplicativo WPF ou um aplicativo de armazenamento do Windows, ele terá um thread dedicado para atualizar a interface do usuário

Exceto pelo Main Thread, todos os threads mencionados até agora são background threads. Ao criar um novo thread, você tem a opção de especificar se o thread deve ser um background threads. Quando o thread principal e todos os outros nonbackground threads de um aplicativo .NET terminam, o aplicativo .NET também termina. Com a chegada dos novos processadores multicore e de vários núcleos, os aplicativos escritos de maneira multithread serão inerentemente beneficiados por essas melhorias, enquanto os aplicativos gravados sequencialmente subutilizam os recursos disponíveis e fazem com que o usuário espere desnecessariamente.

Por padrão, um programa tem um thread chamado Main Thread. O thread principal inicia quando o controle entra no método Main e termina quando o método Main retorna. Um thread pode ser criado usando a classe System.Threading.Thread. Um thread pode ser manipulado apenas em um método. Por exemplo, MainThread precisa de um método Main para controlar o fluxo de um programa. Em um programa C#, um thread pode ser encontrado em qualquer um dos seguintes estados

|  |  |
| --- | --- |
| **Estado** | **Descrição** |
| Não iniciado | O thread foi criado, mas ainda não iniciado |
| Executando | O thread está executando um programa |
| WaitSleepJoin | O thread está bloqueado devido ao método Wait, Sleep ou Join |
| Suspenso | O thread está suspenso |
| Parado | O thread está parado, normalmente ou abortado |

A propriedade ThreadState é útil para fins de diagnóstico, mas inadequada para sincronização, porque o estado de um thread pode mudar entre o teste do ThreadState e a atuação nessas informações.



**Usos e mau uso de thread**

Multithreading tem muitos usos; aqui estão os mais comuns:

* **Manter uma interface de usuário responsiva:** Ao executar tarefas demoradas em um thread paralelo de "trabalhador", o thread principal da interface do usuário fica livre para continuar processando eventos de teclado e mouse.
* **Fazer uso eficiente de uma CPU bloqueada:** O multithreading é útil quando um thread aguarda uma resposta de outro computador ou peça de hardware. Enquanto um thread é bloqueado durante a execução da tarefa, outros threads podem tirar proveito do computador sem carga.
* **Programação paralela:** O código que executa cálculos intensivos pode ser executado mais rapidamente em computadores com vários núcleos ou multiprocessadores se a carga de trabalho for compartilhada entre vários threads em uma estratégia de “dividir e conquistar”.
* **Execução especulativa:** Em máquinas multicore, às vezes você pode melhorar o desempenho prevendo algo que talvez precise ser feito e, em seguida, com antecedência. O LINQPad usa essa técnica para acelerar a criação de novas consultas. Uma variação é executar vários algoritmos diferentes em paralelo que resolvem a mesma tarefa. Qualquer um que terminar primeiro "vence" - isso é eficaz quando você não pode saber antecipadamente qual algoritmo será executado mais rapidamente.
* **Permitir que as solicitações sejam processadas ​​simultaneamente**: Em um servidor, as solicitações do cliente podem chegar simultaneamente e, portanto, precisam ser tratadas em paralelo (o .NET Framework cria threads para isso automaticamente se você usar ASP.NET, WCF, Serviços da Web ou Remoting). Isso também pode ser útil em um cliente (por exemplo, lidar com redes ponto a ponto - ou mesmo várias solicitações do usuário).

Com tecnologias como ASP.NET e WCF, você pode não estar ciente de que o multithreading está ocorrendo - a menos que você acesse dados compartilhados (talvez por meio de campos estáticos) sem o bloqueio apropriado, sem a segurança do thread.

Threads também tem efeitos colaterais. O maior é que o multithreading pode aumentar a complexidade. Ter muitos threads não cria por si só muita complexidade; mas sim a interação entre os threads (normalmente via dados compartilhados). Isso se aplica se a interação é intencional ou não e pode causar longos ciclos de desenvolvimento e uma suscetibilidade contínua a erros intermitentes e não reproduzíveis. Por esse motivo, vale a pena manter a interação no mínimo e seguir projetos simples sempre que possível.

Uma boa estratégia é encapsular a lógica multithreading em classes reutilizáveis ​​que podem ser examinadas e testadas independentemente. O próprio Framework oferece muitas construções de thread de nível superior, abordadas posteriormente.

O thread também incorre em um custo de recurso e CPU na programação e troca de threads (quando há mais threads ativos que núcleos de CPU) - e também há um custo de criação/desmontagem. O multithreading nem sempre acelera seu aplicativo - pode até abrandá-lo se usado em excesso ou de forma inadequada. Por exemplo, quando operação pesada de E/S está envolvida, pode ser mais rápido que alguns threads de trabalho executem tarefas em sequência do que 10 threads sendo executados ao mesmo tempo.

**Gerenciando um thread**

A classe Thread não é algo que você deve usar em seus aplicativos, exceto quando você tiver necessidades especiais. No entanto, ao usar a classe Thread, você tem controle sobre todas as opções de configuração. Você pode, por exemplo, especificar a prioridade do seu Thread, informar ao Windows que seu Thread está demorando ou configurar outras opções avançadas. A classe System.Threading.Thread contém os seguintes métodos e propriedades comuns, que são úteis para gerenciar um thread.

|  |  |
| --- | --- |
| **Propriedades** | **Descrição** |
| [CurrentContext](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.currentcontext?view=netframework-4.8#System_Threading_Thread_CurrentContext) | Obtém o contexto atual no qual o thread está em execução. |
| [CurrentThread](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.currentthread?view=netframework-4.8#System_Threading_Thread_CurrentThread) | Obtém o thread em execução no momento. |
| [IsAlive](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.isalive?view=netframework-4.8#System_Threading_Thread_IsAlive) | Obtém um valor que indica o status de execução do thread atual. |
| [IsBackground](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.isbackground?view=netframework-4.8#System_Threading_Thread_IsBackground) | Obtém ou define um valor que indica se um thread é ou não um thread de segundo plano. |
| [IsThreadPoolThread](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.isthreadpoolthread?view=netframework-4.8#System_Threading_Thread_IsThreadPoolThread) | Obtém um valor que indica se um thread pertence ao pool de threads gerenciados ou não. |
| [ManagedThreadId](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.managedthreadid?view=netframework-4.8#System_Threading_Thread_ManagedThreadId) | Obtém um identificador exclusivo para o thread gerenciado atual. |
| [Name](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.name?view=netframework-4.8#System_Threading_Thread_Name) | Obtém ou define o nome do thread. |
| [Priority](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.priority?view=netframework-4.8#System_Threading_Thread_Priority) | Obtém ou define um valor que indica a prioridade de agendamento de um thread. |
| [ThreadState](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.threadstate?view=netframework-4.8#System_Threading_Thread_ThreadState) | Obtém um valor que contém os estados do thread atual. |

|  |  |
| --- | --- |
| **Métodos** | **Descrição** |
| [Abort()](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.abort?view=netframework-4.8#System_Threading_Thread_Abort) | Gera um [ThreadAbortException](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadabortexception?view=netframework-4.8) no thread no qual ele é invocado, para iniciar o processo de encerramento do thread. Geralmente, a chamada a esse método termina o thread. |
| [Interrupt()](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.interrupt?view=netframework-4.8#System_Threading_Thread_Interrupt) | Interrompe um thread que está no estado de thread [WaitSleepJoin](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadstate?view=netframework-4.8#System_Threading_ThreadState_WaitSleepJoin). |
| [Join()](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.join?view=netframework-4.8#System_Threading_Thread_Join) | Bloqueia o thread de chamada até que o thread representado por essa instância termine, enquanto continua a executar COM padrão e o bombeamento de SendMessage. |
| [Join(Int32)](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.join?view=netframework-4.8#System_Threading_Thread_Join_System_Int32_) | Bloqueia o thread de chamada até que o thread representado por essa instância termine ou até que o tempo especificado tenha decorrido, enquanto continua executando o COM padrão e o bombeamento de SendMessage. |
| [Join(TimeSpan)](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.join?view=netframework-4.8#System_Threading_Thread_Join_System_TimeSpan_) | Bloqueia o thread de chamada até que o thread representado por essa instância termine ou até que o tempo especificado tenha decorrido, enquanto continua executando o COM padrão e o bombeamento de SendMessage. |
| [Resume()](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.resume?view=netframework-4.8#System_Threading_Thread_Resume) | Retoma um thread que foi suspenso. |
| [Sleep(Int32)](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.sleep?view=netframework-4.8#System_Threading_Thread_Sleep_System_Int32_) | Suspende o thread atual no número especificado de milissegundos. |
| [Sleep(TimeSpan)](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.sleep?view=netframework-4.8#System_Threading_Thread_Sleep_System_TimeSpan_) | Suspende o thread atual para o período de tempo especificado. |
| [Start()](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.start?view=netframework-4.8#System_Threading_Thread_Start) | Faz com que o sistema operacional altere o estado da instância atual para [Running](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadstate?view=netframework-4.8#System_Threading_ThreadState_Running). |
| [Start(Object)](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.start?view=netframework-4.8#System_Threading_Thread_Start_System_Object_) | Faz com que o sistema operacional altere o estado da instância atual para [Running](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadstate?view=netframework-4.8#System_Threading_ThreadState_Running) e, opcionalmente, fornece um objeto que contém dados a serem usados pelo método executado pelo thread. |
| [Suspend()](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.suspend?view=netframework-4.8#System_Threading_Thread_Suspend) | Suspende o thread ou, se o thread já está suspenso, não tem efeito. |

**Iniciando um thread**

É um processo simples trabalhar com threads:

1. Crie um objeto thread .
2. Inicie o thread
3. Faça mais trabalho no método de chamada.
4. Aguarde a thread terminar.
5. Continue o trabalho no método de chamada.

Você inicia um thread fornecendo um delegado que representa o método que o thread deve executar em seu construtor de classe. Em seguida, você chama o método [Start](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.start?view=netframework-4.8) para iniciar a execução.

|  |  |
| --- | --- |
| **Construtores** | **Descrição** |
| [Thread(ParameterizedThreadStart)](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.-ctor?view=netframework-4.8#System_Threading_Thread__ctor_System_Threading_ParameterizedThreadStart_) | Inicializa uma nova instância da classe [Thread](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread?view=netframework-4.8), especificando um delegado que permite que um objeto seja passado para o thread quando o thread for iniciado. |
| [Thread(ParameterizedThreadStart, Int32)](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.-ctor?view=netframework-4.8#System_Threading_Thread__ctor_System_Threading_ParameterizedThreadStart_System_Int32_) | Inicializa uma nova instância da classe [Thread](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread?view=netframework-4.8), especificando um delegado que permite que um objeto seja passado para o thread quando o thread é iniciado e especificando o tamanho máximo da pilha para o thread. |
| [Thread(ThreadStart)](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.-ctor?view=netframework-4.8#System_Threading_Thread__ctor_System_Threading_ThreadStart_) | Inicializa uma nova instância da classe [Thread](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread?view=netframework-4.8). |
| [Thread(ThreadStart, Int32)](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.-ctor?view=netframework-4.8#System_Threading_Thread__ctor_System_Threading_ThreadStart_System_Int32_) | Inicializa uma nova instância da classe [Thread](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread?view=netframework-4.8), especificando o tamanho máximo da pilha do thread. |

Dentro do MainThread, um thread pode ser inicializado usando a classe Thread do namespace System.Threading. Um thread pode iniciar sua execução quando um método Thread.Start() é chamado. Os construtores de [Thread](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread?view=netframework-4.8) podem usar um dos dois tipos delegados, dependendo se você pode passar um argumento para o método a ser executado.

[**ThreadStart**](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadstart?view=netframework-4.8)

Se o método não tiver argumentos, você passará um [ThreadStart](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadstart?view=netframework-4.8) é um delegado para o construtor, representa o método que é executado em um thread. Ele tem a assinatura:

public delegate void ThreadStart()

O método Start retorna imediatamente, muitas vezes antes do novo thread realmente começar. Você pode usar as propriedades ThreadState e IsAlive para determinar o estado do thread a qualquer momento, mas essas propriedades nunca devem ser usadas para sincronizar as atividades dos threads.

* Referenciando "MyThreadMethod " usando explicitamente delegate "ThreadStart" sem parâmetros

**Thread variableName = new Thread (novo ThreadStart (MyThreadMethod));**

* Também podemos referenciar "**MyThreadMethod** " para encadear sem usar explicitamente o delegate "ThreadStart".

**Thread variableName = new Thread (MyThreadMethod);**

static void MyThreadMethod()

{

Console.WriteLine("Hello From My Custom Thread");

for (int i = 0; i < 10; i++)

{

Console.Write(" {0} ", i);

}

Console.WriteLine();

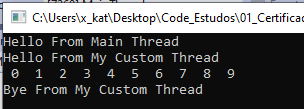
Console.WriteLine("Bye From My Custom Thread");

}

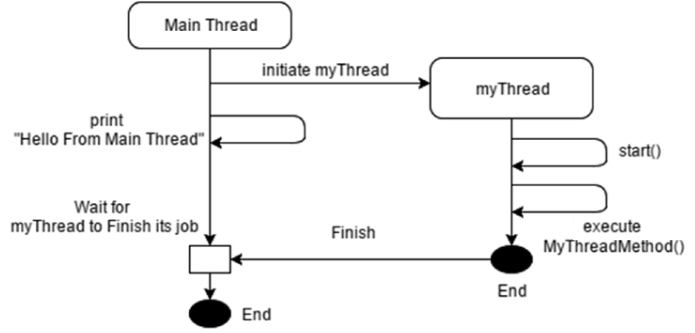
Thread myThread = new Thread(new ThreadStart(MyThreadMethod));

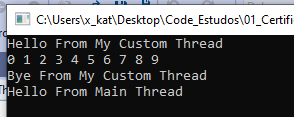
myThread.Start();

Console.WriteLine("Hello From Main Thread");



O Tópico Principal inicializa “mythread” e imprime “Hello From Main Thread”. Enquanto "mythread" estava sendo inicializado, "myThread.Start ()" altera seu estado para execução e depois executa "MyThreadMethod ()". "Hello From Main Thread" fazia parte do MainThread e foi exibido na tela primeiro, porque "myThread" estava demorando para mudar seu estado para execução.





[**ParameterizedThreadStart**](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.parameterizedthreadstart?view=netframework-4.8)

Se o método tiver um argumento, você passará um delegado [ParameterizedThreadStart](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.parameterizedthreadstart?view=netframework-4.8) para o construtor. Esta versão do método MyThreadMethod tem um único parâmetro object. Ele tem a assinatura:

**public delegate void ParameterizedThreadStart(object obj)**

* Exemplo de "MyThreadMethod " usando impicitamente o delegate "ThreadStart" com parâmetros:

static void MyThreadMethod\_Param(Object param)

{

Console.WriteLine("Hello From My Custom Thread");

for (int i = 0; i < (int)param; i++)

{

Console.Write("{0}", i);

}

Console.WriteLine();

Console.WriteLine("Bye From My Custom Thread");

}

Thread myThread = new Thread(MyThreadMethod\_Param);

myThread.Start(20);

//OU

Thread myThread\_param = new Thread(new ParameterizedThreadStart(MyThreadMethod\_Param));

myThread\_param.Start(50);

**Start(Object)**

O Start(Object) faz com que o sistema operacional altere o estado da instância atual para Running e, opcionalmente, fornece um objeto que contém dados a serem usados pelo método executado pelo thread. Quando um thread está no estado de ThreadState.Running, o sistema operacional pode agendá-lo para execução.

public class Work

{

public static void StaticMethod(object data)

{

Console.WriteLine("StaticMethod is running on another thread. Data='{0}'", data);

// Pause for a moment to provide a delay to make threads more apparent.

Thread.Sleep(5000);

Console.WriteLine("StaticMethod called by the worker thread has ended.");

}

public void InstanceMethod(object data)

{

Console.WriteLine("InstanceMethod is running on another thread. Data='{0}'", data);

// Pause for a moment to provide a delay to make threads more apparent.

Thread.Sleep(3000);

Console.WriteLine("InstanceMethod called by the worker thread has ended.");

}

}

// Start a thread that calls a parameterized static method.

Thread newThread = new Thread(Work.StaticMethod);

newThread.Start(42);

Work owork = new Work();

// Start a thread that calls a parameterized instance method.

newThread = new Thread(owork.InstanceMethod);

newThread.Start("The answer.");

**Thread.Join()**

O método Thread.Join() é usado para manter o thread de chamada em espera até que o thread chamado não seja parado ou sua execução seja encerrada (Similiar ao que ocorre ao métodos delegado e a instrução Task.Wait()). Thread.Join() altera o estado do thread de chamada para ThreadState.WaitSleepJoin. Além disso, o Thread.Join() não pode ser chamado em um thread que não esteja no estado ThreadState.Unstarted.

//Instantiate a thread

Thread myThread = new Thread(new ThreadStart(MyThreadMethod));

//Start the execution of thread

myThread.Start();

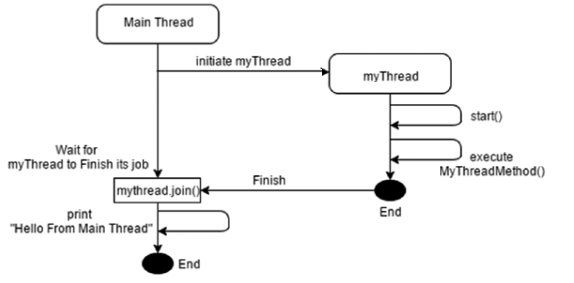
//Wait until mythread is terminated

myThread.Join();

//Everything else is part of Main Thread.

Console.WriteLine("Hello From Main Thread");

Desta vez, devido ao método “mythread.Join()”, MainThread (thread de chamada) imprimirá “Hello From Main Thread” por último, porque o método “mythread.join ()” força MainThread (thread de chamada) a esperar até que mythread seja não encerrado.



**Thread.Sleep**

O Thread.Sleep(5000) é usado para suspender a execução de um thread atual por um número especificado de milissegundos, 5000 milissegundos corresponde a 5 segundos. Thread.Sleep (0) é usado para sinalizar ao Windows que esse thread foi concluído. Em vez de esperar que todo o intervalo de tempo do thread termine, ele mudará imediatamente para outro thread.

O exemplo a seguir usa a sobrecarga do método Sleep(TimeSpan) para bloquear o thread principal do aplicativo cinco vezes, por dois segundos a cada vez.

TimeSpan ts = new TimeSpan(0, 0, 2);

for (int i = 1; i <= 5; i++)

{

Console.WriteLine("Thread paused for {0} second", 2);

// Pause thread for 1 second

Thread.Sleep(ts);

Console.WriteLine("i value: {0}", i);

}

**Multi-Threading**

Para melhorar a capacidade de resposta do seu aplicativo, foi introduzida a noção de multithreading. Multi-threading é a extensão da multitarefa. Multi-threading é a capacidade de um sistema operacional para subdividir a operação específica dentro de um único aplicativo em threads individuais. Multi-threading aumenta a **capacidade** de **resposta** do sistema, pois, se um thread do aplicativo não estiver respondendo, um thread gera outro thread para fazer a busca e a espera enquanto o thread pai continuava executando outro trabalho. Quando os dados eram necessários, o thread pai era bloqueado, aguardando que o thread gerado terminasse seu trabalho e o usuário não precisaria ficar inativo. Esse padrão é conhecido como padrão fork-join.

Em C#, o namespace System.Threading é usado para criar e gerenciar threads em um aplicativo multithread. O multithreading é diferente da multitarefa no sentido de que a multitarefa permite várias tarefas ao mesmo tempo, enquanto o multithread permite que **vários threads de uma única tarefa** (programa, processo) sejam processados ​​pela CPU ao mesmo tempo. Ou seja, o sistema operacional divide o tempo de processamento não apenas entre aplicativos diferentes, mas também entre cada thread dentro de um aplicativo.

|  |  |  |
| --- | --- | --- |
| **Comparação** | **Multitarefa** | **Multithreading** |
| Basic | Permite que a CPU execute várias tarefas ao mesmo tempo. | Permite que a CPU execute vários threads de um processo simultaneamente. |
| Comutação | A CPU alterna entre programas com freqüência. | A CPU alterna entre os threads com freqüência. |
| Memória e Recurso | O sistema tem que alocar memória e recursos separados para cada programa que a CPU está executando. | O sistema tem que alocar memória para um processo, vários threads desse processo compartilha a mesma memória e recursos alocados para o processo |

A seguir, estão algumas das desvantagens dos aplicativos multithread:

* Todos os threads consomem muitos recursos. Eles precisam de muita memória (1 MB é padrão) e sempre que o agendador alternar entre os threads, o processador estará ocupado salvando o contexto do thread de suspensão e restaurando o contexto do thread em execução.
* Se seu aplicativo criar muitos threads, a alternância de contexto consumirá uma quantidade considerável de tempo.
* Como o thread precisa de muita memória, geralmente leva uma quantidade considerável de tempo para o sistema criar uma banda de rodagem e também leva algum tempo para destruí-la.

Quando um processo é iniciado, o Common Language Runtime cria automaticamente um única thread de primeiro plano para executar o código do aplicativo. Junto com esse thread de primeiro plano principal, um processo pode criar um ou mais threads para executar uma parte do código do programa associado ao processo. Esses threads podem ser executados em primeiro plano ou em segundo plano. Além disso, você pode usar a classe [**ThreadPool**](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadpool?view=netframework-4.8) para executar código em threads de trabalho que são gerenciados pelo Common Language Runtime.

* A thread única, também chamado de thread 'principal', é criado automaticamente pelo sistema operacional.
* Uma thread é um caminho de execução e é executada no ambiente isolado do processo. Ou seja, é executada dentro de um processo separado do sistema operacional e opera independentemente de outros caminhos de execução
* Outra thread pode ser criada e iniciada:
  + Usando a classe Thread
  + Chamando o método Start

static void RunSequencial()

{

double result = 0d;

// Call the function to read data from I/O

result += ReadDataFromIO();

// Add the result of the second calculation

result += DoIntensiveCalculations();

// Print the result

Console.WriteLine("The result is {0}", result);

}

static void RunWithThreads()

{

double result = 0d;

// Create the thread to read from I/O

var thread = new Thread(() => result = ReadDataFromIO());

// Start the thread

thread.Start();

// Save the result of the calculation into another variable

double result2 = DoIntensiveCalculations();

// Wait for the thread to finish

thread.Join();

// Calculate the end result

result += result2;

// Print the result

Console.WriteLine("The result is {0}", result);

}

static double ReadDataFromIO()

{

// We are simulating an I/O by putting the current thread to sleep.

Thread.Sleep(5000);

return 10d;

}

static double DoIntensiveCalculations()

{

// We are simulating intensive calculations

// by doing nonsens divisions

double result = 100000000d;

var maxValue = Int32.MaxValue;

for (int i = 1; i < maxValue; i++)

{

result /= i;

}

return result + 10d;

}

// We are using Stopwatch to time the code

Stopwatch sw = Stopwatch.StartNew();

// Here we call different methods

// for different ways of running our application.

RunSequencial();

RunWithThreads();

// Print the time it took to run the application.

Console.WriteLine("We're done in {0}ms!", sw.ElapsedMilliseconds);

if (Debugger.IsAttached)

{

Console.Write("Press any key to continue . . .");

Console.ReadKey(true);

}

**ThreadPriority**

A prioridade da thread define quanto tempo de CPU um thread terá para execução. Quando um thread é criado, inicialmente ele é atribuído com prioridade Normal. Tanto o seu processo como o seu thread têm uma prioridade. A atribuição de baixa prioridade é útil para aplicativos como um protetor de tela. Esse aplicativo não deve competir com outros aplicativos pelo tempo de CPU. Um thread de prioridade mais alta deve ser usado apenas quando for absolutamente necessário. Um thread pode ser atribuído com qualquer uma das seguintes prioridades:

|  |  |
| --- | --- |
| **Prioridade** | **Explicação** |
| High | O thread agendará antes dos threads com qualquer prioridade |
| AboveNormal | O thread será agendado antes dos threads com prioridade normal |
| Normal | Será agendado antes dos Threads com prioridade BelowNormal |
| BelowNormal | O thread será agendado antes dos threads com prioridade mais baixa |
| Lowest | Agendará após Threads com prioridade BelowNormal |

private static void myMethod()

{

//Get Name of Current Thread

string threadName = Thread.CurrentThread.Name.ToString();

//Get Priority of Current Thread

string threadPriority = Thread.CurrentThread.Priority.ToString();

uint count = 0;

while (stop != true) { count++; }

Console.WriteLine("{0,-11} with {1,11} priority has a count = {2,13}",

Thread.CurrentThread.Name, Thread.CurrentThread.Priority.ToString(), count);

}

Thread thread1 = new Thread(new ThreadStart(myMethod));

thread1.Name = "Thread 1";

thread1.Priority = ThreadPriority.Lowest;

Thread thread2 = new Thread(new ThreadStart(myMethod));

thread2.Name = "Thread 2";

thread2.Priority = ThreadPriority.Highest;

Thread thread3 = new Thread(new ThreadStart(myMethod));

thread3.Name = "Thread 3";

thread3.Priority = ThreadPriority.BelowNormal;

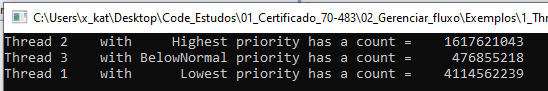
thread1.Start();

thread2.Start();

thread3.Start();

Thread.Sleep(10000);

stop = true;



O exemplo acima mostra que o tempo de CPU de um thread depende de sua prioridade. No exemplo, o Thread 2 tem uma prioridade acima dos outros, portanto, incrementa mais o valor da contagem usando mais tempo de CPU. Embora o Thread 3 tenha a menor prioridade, portanto, o número de incrementos é menor que os outros threads.

**ThreadStatic**

ThreadStatic é um atributo usado no topo de um campo estático para tornar seu valor exclusivo (local) para cada thread.

[ThreadStatic]

static int \_count\_ts = 0;

static int \_count = 0;

static void Main(string[] args)

{

Thread threadA = new Thread(() =>

{

for (int i = 0; i < 10; i++)

{

Console.WriteLine("ThreadA \_count\_ts = {0} ", \_count\_ts++);

Console.WriteLine(" ThreadA \_count = {0} ", \_count++);

}

});

Thread threadB = new Thread(() =>

{

for (int i = 0; i < 10; i++)

{

Console.WriteLine("ThreadB \_count\_ts = {0} ", \_count\_ts++);

Console.WriteLine(" ThreadB \_count = {0} ", \_count++);

}

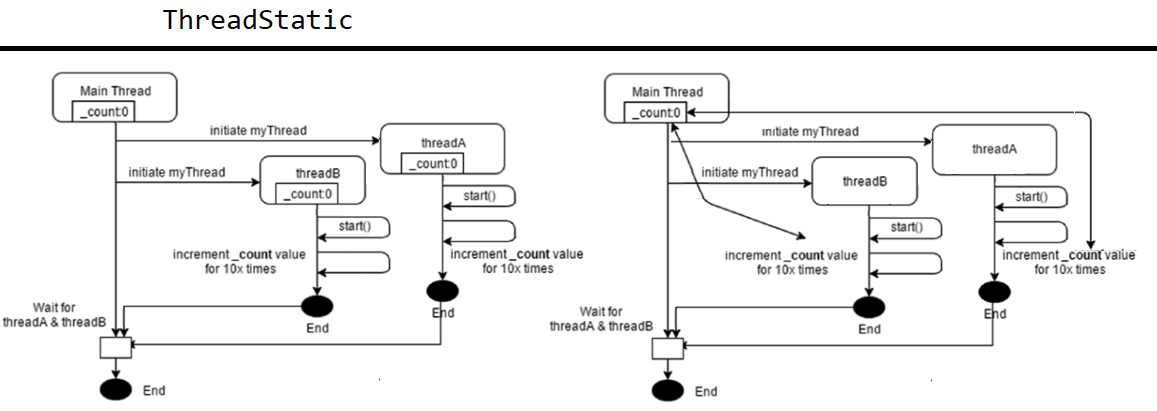
});

threadA.Start();

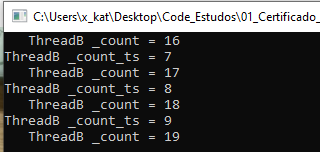
threadB.Start();

Console.ReadKey();

}



No snippet de código acima, o threadA e o threadB têm seus valores locais exclusivos com o atributo "ThreadStatic" \_count\_ts. Ambos os threads aumentaram o valor de \_count 10 vezes. O resultado final não é 19, porque cada thread aumentou o valor de sua cópia local da variável \_count\_ts. Na variável \_count não marcou com o atributo "ThreadStatic", portanto, os dois threads compartilharam a mesma variável \_count. Quando um thread incrementa o valor de \_count, ele afeta o valor da \_count que é usada no outro thread chegando o valor até 19.



**Classe ThreadLocal<T>**

Se você deseja usar dados locais em um thread e inicializá-lo para cada thread, pode usar a classe ThreadLocal <T>. Essa classe leva um delegado para um método que inicializa o valor. O código abaixo mostra um exemplo.

public static ThreadLocal<int> \_field = new ThreadLocal<int>(() =>

{

return Thread.CurrentThread.ManagedThreadId;

});

static void Main(string[] args)

{

new Thread(() =>

{

for (int x = 0; x < \_field.Value; x++)

{

Console.WriteLine("Thread A: {0}", x);

}

}).Start();

new Thread(() =>

{

for (int x = 0; x < \_field.Value; x++)

{

Console.WriteLine("Thread B: {0}", x);

}

}).Start();

Console.ReadKey();

}

Aqui você vê outro recurso do .NET Framework. Você pode usar a classe Thread.CurrentThread para solicitar informações sobre o thread que está sendo executado. Isso é chamado de contexto de execução do thread. Essa propriedade fornece acesso a propriedades como a cultura atual do thread (um CultureInfo associado ao thread atual usado para formatar datas, horas, números, valores de moeda, a ordem de classificação do texto, convenções de maiúsculas e minúsculas) e principal ( representando o contexto de segurança atual), prioridade (um valor para indicar como o thread deve ser agendado pelo sistema operacional) e outras informações.

Quando um thread é criado, o tempo de execução garante que o contexto de execução do thread inicial flua para o novo thread. Dessa forma, o novo thread tem os mesmos privilégios que o thread pai.

Essa cópia de dados custa alguns recursos, no entanto. Se você não precisar desses dados, poderá desativar esse comportamento usando o método ExecutionContext.SuppressFlow.

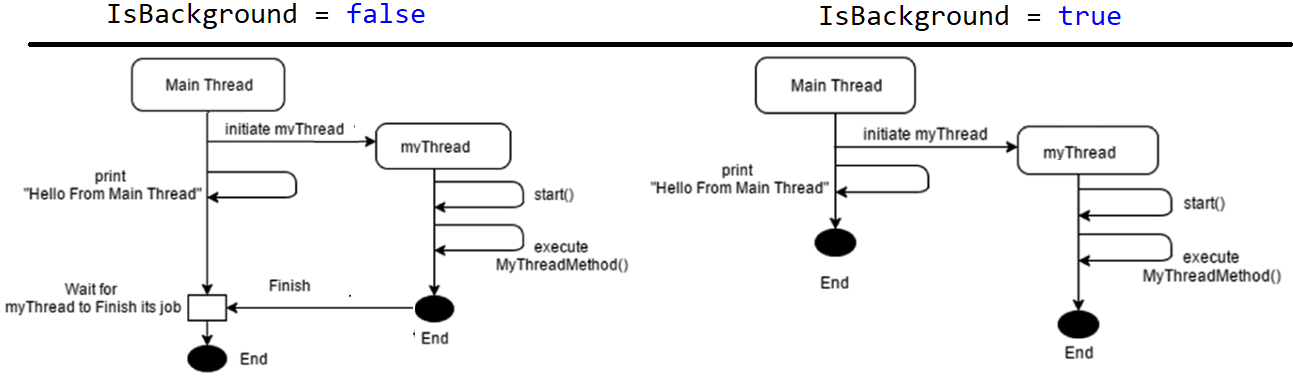
**Threads em primeiro e segundo plano**

Threads em segundo plano são idênticos aos threads em primeiro plano com uma exceção: um thread em segundo plano não mantém um processo em execução se todos os threads de primeiro plano tiverem terminado. Depois que todos os threads de primeiro plano forem interrompidos, o tempo de execução interromperá todos os threads em segundo plano e será desligado. As instâncias da classe [Thread](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread?view=netframework-4.8) representam threads em:

* **Primeiro plano (Foreground)**
* O thread do aplicativo principal.
* Todos os threads criados chamando um construtor de classe [Thread](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread?view=netframework-4.8).
* Mantem o aplicativo ativo enquanto estiver em execução
* Quando concluído, o aplicativo termina e todos os threads em segundo plano são encerrados imediatamente
* Por padrão, threads criados explicitamente são threads em primeiro plano
* **Segundo plano (Background)**
* Threads do pool de threads mantidos pelo tempo de execução. Você pode configurar o pool de threads e agendar o trabalho em threads do pool de threads usando a classe [ThreadPool](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadpool?view=netframework-4.8).
* As operações assíncronas que usam as classes [Task](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.tasks.task?view=netframework-4.8) e [Task<TResult>](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.tasks.task-1?view=netframework-4.8) baseadas em tarefas são executadas automaticamente em threads do pool de threads.
* Todos os threads que entram no ambiente de execução gerenciado por meio de código não gerenciado.
* Excelente para tarefas como serviços de pesquisa ou informações de registro

Os threads em segundo plano são úteis para qualquer operação que deve continuar desde que um aplicativo esteja em execução, mas não deve impedir que o aplicativo seja encerrado, como monitoramento de alterações do sistema de arquivos ou conexões de soquete de entrada.

Você pode alterar um thread para ser executado em segundo plano definindo a propriedade [IsBackground](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.isbackground?view=netframework-4.8) a qualquer momento. Por padrão, o valor de Thread.IsBackground é falso, que o torna um thread em primeiro plano.



Se você executar o código a seguir com a propriedade IsBackground definida como true, o aplicativo será encerrado imediatamente. Se você defini-lo como false (criando um thread em primeiro plano), o aplicativo imprimirá a mensagem ThreadProc dez vezes.

static void MyThreadMethod()

{

Console.WriteLine("Start of MyThread");

for (int i = 0; i < 10; i++)

{

//suspend the thread for 100 milliseconds

Thread.Sleep(100);

Console.Write("{0} ", i);

}

Console.WriteLine();

Console.WriteLine("End of MyThread");

Console.ReadKey();

}

//Instantiate a thread

Thread myThread = new Thread(new ThreadStart(MyThreadMethod));

//by default Isbackgrount value is always false

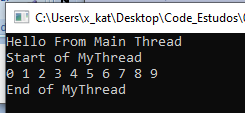
myThread.IsBackground = false;

//Start the execution of thread

myThread.Start();

//Everything else is part of Main Thread.

Console.WriteLine("Hello From Main Thread");



O exemplo a seguir ilustra a diferença entre os threads de primeiro plano e de segundo plano sem utilizar a propriedade IsBackground. O método Task.Run é novo no .NET Framework 4.5, ele inicia uma tarefa que é executada em um thread em segundo plano a partir do ThreadPool. Como mostra a saída, o loop é interrompido antes de ser executado por cinco segundos. O Thread.Sleep(500) é usado para suspender a execução de um thread atual por um número especificado de milissegundos.

static void Count()

{

// Set this condition number higher than 8 (which is the condition in Task.Run)

// to see that this foreground thread continues to run when the Task completes

// Set this condition number lower than 8 to see that when this foreground thread

// completes the background task thread ends.

for (int i = 0; i < 4; i++)

{

System.Threading.Thread.Sleep(500);

Console.Write("FG ");

}

}

// Starts a new foreground thread running the Count method

// writes "FG" in the output

Thread t = new Thread(Count);

t.Start();

// Starts a background thread that writes "BG" in the output

Task task = Task.Run(() =>

{

// The task is running on a background thread and will

// produce 8 BG's in the output

for (int i = 0; i < 8; i++)

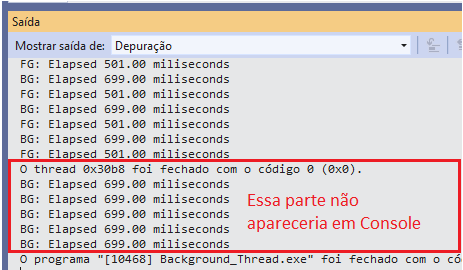
{

Thread.Sleep(500);

Console.Write("BG ");

}

});



**Classe BackgroundWorker**

O .NET 2.0 introduziu uma classe chamada System.ComponentModel.BackgroundWorker que abstrai a criação do thread e o uso do pool de threads. A implementação da classe BackgroundWork garante que, em um aplicativo Windows Forms ou WPF, o manipulador de eventos RunWorkerCompleted seja executado por um thread da interface do usuário se RunWorkerAsync for chamado pelo thread da interface do usuário. Em outras palavras, se você iniciar o trabalho em segundo plano dentro de um manipulador de eventos, o evento de conclusão será executado no thread da interface do usuário.

As tabelas a seguir mostram alguns dos métodos, propriedades e eventos que essa classe oferece.

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| CancellationPending | Defina como true se CancelAsync foi chamado para esta operação em segundo plano. |
| IsBusy | Retorna true depois que o RunAsync foi chamado e antes da operação em segundo plano ser concluída. |
| WorkerReportsProgress | Defina essa propriedade como true se desejar que sua operação em segundo plano relate atualizações de progresso. |
| WorkerSupportsCancellation | Defina essa propriedade como true se desejar que sua operação em segundo plano suporte o cancelamento. |

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| RunWorkerAsync | Registra uma solicitação de início para a operação em segundo plano |
| ReportProgress | Gera o evento ProgressChanged |
| CancelAsync | Registra uma solicitação de cancelamento para a operação em segundo plano. |

|  |  |
| --- | --- |
| **Evento** | **Descrição** |
| DoWork | Dispara quando RunWorkerAsync é chamado. É aqui que você chama seu método de execução longa. |
| ProgressChanged | Dispara quando ReportProgress é chamado. |
| RunWorkerCompleted | Dispara quando a operação em segundo plano é concluída. Isso pode ser feito porque a operação foi concluída com êxito, como resposta a uma solicitação de cancelamento ou devido a uma exceção não tratada |

O fluxo de trabalho do uso da classe BackgroundWorker é o seguinte:

1. Crie um método que segue a assinatura DoWorkEventHandler.
2. Nesse método, chame a operação de execução longa. Quando a operação terminar, atribua o resultado da operação à propriedade Result do parâmetro DoWorkEventArgs.
3. Crie uma instância do BackgroundWorker.
4. Use o método que você criou na primeira etapa para se inscrever no evento DoWork.
5. Crie um método que segue a assinatura RunWorkerCompletedEventHandler.
6. Nesse método, obtenha o resultado da operação de longa execução e atualize a interface do usuário.
7. Use este método para assinar o evento RunWorkerCompleted, para que seu código saiba quando a operação de longa duração for concluída. Antes de ler o resultado, verifique se a operação de longa execução não gerou uma exceção verificando a propriedade Error do parâmetro RunWorkerCompletedEventArgs. Se a propriedade for nula, significa que nenhuma exceção foi lançada.
8. Opcionalmente, crie um método a ser usado para relatar o progresso, seguindo a assinatura ProgressChangedEventHandler e inscrevendo esse método no evento ProgressChanged.
9. Chame o método RunWorkerAsync para iniciar o trabalho em segundo plano.
10. Se o trabalho oferecer suporte ao cancelamento e você quiser cancelar o trabalho, poderá chamar o método CancelAsync.

Suponha que você tenha um aplicativo Windows Forms que tenha um formulário com um rótulo chamado lblResult e um botão chamado btnRun. Se você usar o método de longa duração da seção anterior juntamente com BackgroundWork, o código resultante pode ter esta aparência

private void btnRun\_Click(object sender, EventArgs e)

{

\_worker = new BackgroundWorker();

\_worker.DoWork += \_worker\_DoWork;

\_worker.RunWorkerCompleted += \_worker\_RunWorkerCompleted;

if (!\_worker.IsBusy)

{

\_worker.RunWorkerAsync();

//ERRO de race conditions

//new Thread(() => \_worker.RunWorkerAsync()) { Name = "RunWorkThread" }.Start();

}

}

//ERRO de race conditions

//void \_worker\_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)

//{

// lblResult.Text = e.Result.ToString();

//}

void \_worker\_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)

{

if (this.InvokeRequired)

this.Invoke(new Action<string>(UpdateLabel), e.Result.ToString());

else

UpdateLabel(e.Result.ToString());

}

private void UpdateLabel(string text)

{

lblResult.Text = text;

}

void \_worker\_DoWork(object sender, DoWorkEventArgs e)

{

// Ocorre erro pois está tentando atribuir o valor de lblResult.Text

// Que esta no thread da interface (primeiro plano) que chamou o BackgroundWorker

// para realizar a tarefa que está em outro thread em segundo plano

//lblResult.Text = DoIntensiveCalculations().ToString();

e.Result = DoIntensiveCalculations();

}

static double DoIntensiveCalculations()

{

// We are simulating intensive calculations

// by doing nonsens divisions

double result = 100000000d;

var maxValue = Int32.MaxValue;

for (int i = 1; i < maxValue; i++)

{

result /= i;

}

return result + 10d;

}

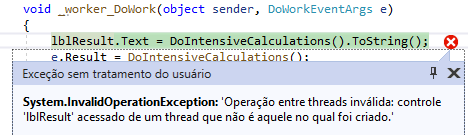
Observe que:

1. Você moveu o código da interface do usuário para seu próprio método. Nesse caso, você chamou o método UpdateLabel.
2. No método worker\_RunWorkerCompleted, você verifica a propriedade InvokeRequired. Essa propriedade é definida na classe Control e, como tal, está presente em todos os controles em uma página. InvokeRequired é definido como false se você o chamar do thread da interface do usuário e true, caso contrário.
3. Se você estiver no thread da interface do usuário, basta chamar o UpdateLabel. Se você estiver em outro thread, deverá chamar o método Invoke, que é definido na classe Control.
4. O método Invoke toma como o primeiro parâmetro um delegado, significando que qualquer método pode ser colocado lá. A nova chamada do construtor Action<string> () é usada para garantir que você obtenha um delegado. Se o seu método tiver uma assinatura diferente, você deverá alterar esse construtor adequadamente. O restante dos parâmetros do método Invoke são enviados diretamente para o método que você deseja executar. Invoke coloca a chamada do método em uma fila para ser atendida pelo thread da interface do usuário.

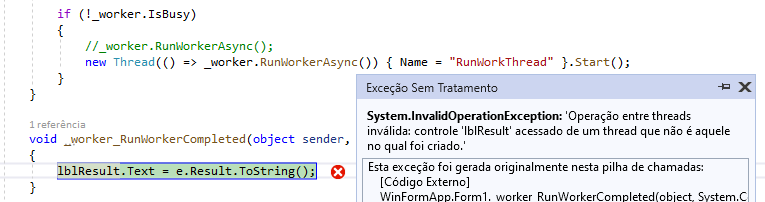
**Race Conditions**

Um dos maiores problemas mencionados anteriormente foi que seu aplicativo pode não responder, dando ao usuário a impressão de que o aplicativo está desligado. O motivo disso geralmente ocorre porque um trabalho pesado é colocado no thread responsável pela atualização da interface do usuário. Para melhorar o desempenho percebido do aplicativo, você pode mover esse trabalho pesado para outro thread.

Os aplicativos Windows Forms e WPF têm threads dedicados que atualizam a interface do usuário para evitar uma situação que possa surgir em aplicativos multithread, chamados condições de corrida (race conditions). Uma condição de corrida ocorre quando dois ou mais threads acessam dados compartilhados, para gravação, ao mesmo tempo. Se você tentar atualizar a interface do usuário (thread em primeiro plano) de outro thread (que realiza o serviço, em segundo plano), o .NET Framework emitirá uma InvalidOperationException contendo a seguinte mensagem:



Se o BackgroundWorker for iniciado a partir de outra thread a partir da interface do usuário, você obteria a mesma exceção mencionada anteriormente.



Como nos aplicativos Windows Forms, os aplicativos WPF têm um thread de interface do usuário dedicado. Diferentemente do Windows Forms, o WPF possui um thread extra responsável pela renderização da interface do usuário. Esse segundo thread é privado para o WPF e você não tem acesso a ele no seu aplicativo. O código para o aplicativo WPF parece quase o mesmo do aplicativo Windows Forms, a única diferença é como você atualiza o rótulo - em vez de definir a propriedade Text, você define a propriedade Content. Esta solução sofre do mesmo problema que a solução Windows Forms sofreu. Se o BackgroundWorker for acionado a partir de outro thread que não seja o thread da interface do usuário, quando você tentar atualizar a interface do usuário, o .NET Framework emitirá uma InvalidOperationException. A solução para esse problema é semelhante à da seção anterior, mas é muito mais simples. O worker\_RunWorkerCompleted deve se parecer com o seguinte:

void \_worker\_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e) {

// lblResult.Content = e.Result;

// Instead of updating the UI directly we call Dispatcher.Invoke

this.Dispatcher.Invoke(() => lblResult.Content = e.Result);

}

Como você pode ver, basta chamar o método Dispatcher.Invoke em todas as situações. Essa chamada garante que a expressão lambda é executado pelo thread da interface do usuário, independentemente de qual thread o método é chamado. Quando você precisar atualizar a interface do usuário, mantenha o trabalho que não seja da interface do usuário ao mínimo. Você deve despachar apenas o código que atualiza a interface do usuário (no exemplo anterior, apenas o código que atualiza o rótulo). Se no seu código, depois de obter o resultado do BackgroundWorker, você precisar fazer mais processamento antes de atualizar a interface do usuário, faça esse processamento antes de chamar o método Dispatch.Invoke para aplicativos WPF ou o método Invoke para aplicativos Windows Forms.

Se o seu aplicativo for direcionado para o .NET 4.5, o novo modelo de programação async/await cuida disso para você. Se você direcionar uma versão do .NET anterior à 4.5, deverá cuidar disso sozinho.

**THREADPOOL**

O ThreadPool é uma coleção de threads recicláveis em segundo plano pré-criados por um sistema e estão disponíveis para executar qualquer tarefa quando necessário. Iniciar um novo thread requer recursos (Normalmente algumas centenas de microssegundos), isso pode se tornar visível se muitos threads estiverem sendo criados para operações curtas. Muitos aplicativos também criam threads que passam muito tempo no estado de repouso, aguardando a ocorrência de um evento. Existem situações também em que threads podem entrar em estado de suspensão apenas para serem despertados periodicamente para sondar informações de status de alteração ou atualização.

O pool de threads permite que você use threads com mais eficiência em tais situações, fornecendo ao aplicativo um pool de threads de trabalho que são gerenciados pelo sistema. Quando um programa requer um thread extra, é mais eficiente usar os threads livre disponíveis em na coleção de threads do ThreadPool, pois pode economizar o custo de criação de um thread. Quando um thread conclui sua execução, ele pode voltar ao ThreadPool para que outros programas possam reutilizar o mesmo thread novamente.

Quando o pool de threads reutiliza um thread, ele não limpa os dados no armazenamento local de thread ou em campos que são marcados com o atributo [**ThreadStaticAttribute**](https://docs.microsoft.com/pt-br/dotnet/api/system.threadstaticattribute?view=netframework-4.8) ([ThreadStatic] visto anteriormente). Portanto, quando um método examina o armazenamento local do thread ou os campos marcados com o atributo ThreadStatic, os valores que ele encontra podem ser deixados de um uso anterior do thread do pool de threads.

Há um pool de threads por processo. A partir do .NET Framework 4, o tamanho padrão do pool de threads de um processo depende de vários fatores, como o tamanho do espaço de endereço virtual. O pool de threads cria e destrói threads de trabalho a fim de otimizar a taxa de transferência, que é definida como o número de tarefas concluídas por unidade de tempo. Pouquíssimos threads podem não fazer um uso ideal dos recursos disponíveis, enquanto muitos threads podem aumentar a contenção de recursos. Como o ThreadPool limita o número disponível de threads, você obtém um grau menor de paralelismo do que a classe Thread comum. Mas o pool de threads também tem muitas vantagens.

Por exemplo, um servidor Web que atenda solicitações recebidas. Todos esses pedidos chegam em um tempo e frequência desconhecidos. O conjunto de threads garante que cada solicitação seja adicionada à fila e que, quando um thread se torna disponível, é processado. Isso garante que seu servidor não trava com a quantidade de solicitações. Se você estender threads manualmente, poderá facilmente derrubar seu servidor se receber muitas solicitações. Cada solicitação possui características únicas no trabalho que elas precisam fazer. O que o pool de threads faz é mapear esse trabalho nos threads disponíveis no sistema. Obviamente, você ainda pode receber tantas solicitações que fica sem threads. Os pedidos começam a ficar na fila e isso faz com que o servidor da Web não responda.

O pool de threads gerencia automaticamente a quantidade de threads necessária para manter-se por perto. Quando é criado, ele começa vazio. Quando uma solicitação é recebida, ele cria threads adicionais para lidar com essas solicitações. Desde que ela possa concluir uma operação antes que uma nova chegue, nenhum novo thread precisa ser criado. Se novos threads não estiverem mais em uso após algum tempo, o conjunto de threads poderá eliminar esses threads, para que não usem mais nenhum recurso.

A classe System.Threading.ThreadPool oferece vários métodos estáticos que você pode chamar para manipular o número de threads no pool de threads que podem ser usados para executar tarefas, postar os itens de trabalho, processar E/S assíncrona, aguardar em nome de outros threads e processar temporizadores. A tabela abaixo lista alguns dos métodos que poedem ser utilizados.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| CorSetMaxThreads | Função definida no arquivo mscoree.h. Para alterar o número de threads de código não gerenciado |
| GetAvailableThreads | Recupera a diferença entre o número máximo de threads do pool de threads retornados pelo método [GetMaxThreads](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadpool.getmaxthreads?view=netframework-4.8#System_Threading_ThreadPool_GetMaxThreads_System_Int32__System_Int32__) e o número de ativos no momento. Isso representa o número de threads que podem selecionar itens de trabalho da fila. |
| [GetMaxThreads](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadpool.getmaxthreads?view=netframework-4.8) | Recupera o número real máximo de threads que podem ser criados pelo pool de threads |
| [GetMinThreads](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadpool.getminthreads?view=netframework-4.8) | Recupera o número real mínimosde threads que estarão disponíveis no pool de threads |
| QueueUserWorkItem | Adiciona uma solicitação de execução à fila do conjunto de threads. Se houver threads disponíveis no pool de threads, a solicitação será executada imediatamente. |
| [SetMaxThreads](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadpool.setmaxthreads?view=netframework-4.8) | Define o número máximo de threads que podem ser criados no pool de threads. |
| [SetMinThreads](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadpool.setminthreads?view=netframework-4.8) | Define o número mínimo de threads que estarão disponíveis no conjunto de threads a qualquer momento. |
| RegisterWaitForSingleObject | Registra um método a ser chamado quando o WaitHandle é especificado quando o primeiro parâmetro é sinalizado ou quando o tempo limite especificado como quarto parâmetro passa. Esse método possui quatro sobrecargas, uma para cada modo, que o tempo limite pode ser expresso como: int, long, unsigned int ou TimeSpan. |

Essa lista não está completa e é recomendado não utilizar nenhum desses métodos. Dessa lista, é mais provável que você use apenas um método: QueueUserWorkItem. O ThreadPool terá um desempenho melhor com seu próprio algoritmo de alocação de threads. Tocar com os limites do conjunto de threads geralmente resulta em desempenho pior, não melhor. Se você acha que seu aplicativo precisa de centenas ou milhares de threads, há algo seriamente errado com a arquitetura do aplicativo e a maneira como ele está usando threads. O método QueueUserWorkItem tem duas sobrecargas:

* public static bool QueueUserWorkItem (WaitCallback callBack)
* public static bool QueueUserWorkItem (WaitCallback callBack, Object state)

O parâmetro do tipo System.Threading.WaitCallback, é um delegado definido da seguinte forma:

* public delegate void WaitCallback(Object state)

O pool de threads funciona da seguinte maneira. Quando você precisa que um método de execução longa seja executado em um thread separado, em vez de criar um novo thread, chame o método QueueUserWorkItem para colocar um novo item de trabalho em uma fila gerenciada pelo pool de threads. Não é possível cancelar um item de trabalho após ele ter sido enfileirado. Se houver um thread inativo no pool, ele seleciona o item de trabalho e o executa até a conclusão, como qualquer thread. Se não houver um thread disponível e o número total no pool for menor que MaxThreads, o pool criará um novo thread para executar o item de trabalho; caso contrário, o item de trabalho do trabalho aguardará na fila pelo primeiro thread disponível. SetMinThread é usado para preencher previamente o pool com threads para melhorar o desempenho do seu aplicativo quando você souber que irá usar o pool de threads. A prioridade de um thread em pool pode ser alterada (Thread.CurrentThread.Priority = ThreadPriority.AboveNormal) , e ele será restaurado ao normal quando liberado de volta para o pool

static void ThreadProc(Object stateInfo)

{

// No state object was passed to QueueUserWorkItem, so stateInfo is null.

Console.WriteLine("Hello from the thread pool.");

}

static double ReadDataFromIO()

{

// We are simulating an I/O by putting the current thread to sleep.

Thread.Sleep(5000);

return 10d;

}

// Queue the thread.

ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadProc));

Console.WriteLine("Hello From Main Thread.");

// The thread pool uses background threads, its important

// to keep main thread busy otherwise program will terminate

// before the background thread complete its execution

Console.ReadLine(); //Wait for Enter

Console.WriteLine("Hello Again from Main Thread.");

// Queue the thread with Lambda

ThreadPool.QueueUserWorkItem((s) =>

{

Console.WriteLine("Hi I'm another free thread from thread pool");

});

ThreadPool.QueueUserWorkItem((x) => result += ReadDataFromIO());

// Save the result of the calculation into another variable

double result2 = 30;

result += result2;

// Print the result

Console.WriteLine("The result is {0}", result);

Console.ReadLine(); //Wait for Enter

**Explicação**

* WaitCallback é um delegado que representa um método de retorno de chamada a ser executado por um thread pool thread. O método que ele representa pega um objeto.
* ThreadPool.QueueUserWorkItem enfileira um thread em segundo plano disponível para execução.
* Console.ReadLine mantém a thread principal em espera até o usuário pressionar “Enter”.

**Limitação do** [**ThreadPool**](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadpool?view=netframework-4.8)

1. É difícil saber quando um thread de um pool de threads terminou sua execução.
2. Não existe um método “Start”, portanto, não podemos saber quando um thread de um pool de threads iniciou sua execução porque está sendo gerenciado pelo sistema.

* Não é possível abortar ou interromper um thread do pool de threads. E o bloqueio de threads em pool pode prejudicar o desempenho
* Você não pode definir o nome de um thread em pool, tornando a depuração mais difícil
* Threads em pool são sempre threads em segundo plano. Ou seja, suas propriedades [IsBackground](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.thread.isbackground?view=netframework-4.8) são true. Isso significa que um thread [ThreadPool](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadpool?view=netframework-4.8) não manterá um aplicativo em execução depois que todos os threads de primeiro plano tiverem saído.
* Não é possível utiliar Join em um thread do pool de threads. Para conseguir isso, você deve usar alguns outros mecanismos.
* Não é possível gerenciar um thread que retorna um valor.

**Exemplos de operações que usam** [**ThreadPool**](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.threadpool?view=netframework-4.8)**:**

* Quando você cria um objeto [Task](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.tasks.task?view=netframework-4.8) ou [Task<TResult>](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.tasks.task-1?view=netframework-4.8) para executar alguma tarefa de forma assíncrona, por padrão, a tarefa é agendada para ser executada em um thread do pool de threads.
* Temporizadores assíncronos usam o pool de threads. Threads de pool de threads executam retornos de chamada da classe [System.Threading.Timer](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.timer?view=netframework-4.8) e geram eventos da classe [System.Timers.Timer](https://docs.microsoft.com/pt-br/dotnet/api/system.timers.timer?view=netframework-4.8).
* Quando você usa identificadores de espera registrados, um thread do sistema monitora o status dos identificadores de espera. Quando uma operação de espera é concluída, um thread de trabalho do pool de threads executa a função de retorno de chamada correspondente.

**Delegados assíncronos**

O ThreadPool.QueueUserWorkItem não fornece um mecanismo fácil para recuperar valores de retorno de um thread após a conclusão da execução. Invocações de delegado assíncronas (delegados assíncronos para breve) resolvem isso, permitindo que qualquer número de argumentos digitados seja transmitido nas duas direções. Além disso, as exceções não tratadas nos delegados assíncronos são reconvertidas no thread original (ou mais precisamente, no thread que chama EndInvoke) e, portanto, não precisam de tratamento explícito.

Veja como você inicia uma tarefa de trabalho por meio de um delegado assíncrono:

1. Instancie um delegado visando o método que você deseja executar em paralelo (normalmente um dos delegados predefinidos do Func).
2. Chame BeginInvoke no delegado, salvando seu valor de retorno IAsyncResult.
   1. BeginInvoke retorna imediatamente para o chamador. Você pode executar outras atividades enquanto o thread em pool estiver funcionando.
3. Quando você precisar dos resultados, chame EndInvoke no delegado, passando o objeto IAsyncResult salvo.

No exemplo a seguir, usamos uma chamada de representante assíncrona para executar simultaneamente com o thread principal, um método simples que retorna o comprimento de uma string:

static int Work(string s) { return s.Length; }

Func<string, int> method = Work;

IAsyncResult cookie = method.BeginInvoke("test", null, null);

//

// ... here's where we can do other work in parallel...

//

int result = method.EndInvoke(cookie);

Console.WriteLine("String length is: " + result);

EndInvoke faz três coisas

1. Aguarda o delegado assíncrono concluir a execução, se ainda não o tiver feito.
2. Ele recebe o valor de retorno (assim como quaisquer parâmetros ref ou out).
3. Lança qualquer exceção de trabalhador não tratado de volta ao thread de chamada.

Se o método que você está chamando com um delegado assíncrono não tiver valor de retorno, você ainda (tecnicamente) é obrigado a chamar o EndInvoke. Se você optar por não chamar para o EndInvoke, no entanto, precisará considerar o tratamento de exceções no método worker para evitar falhas silenciosas.

Você também pode especificar um delegado de retorno de chamada ao chamar BeginInvoke - um método que aceita um objeto IAsyncResult que é chamado automaticamente após a conclusão. Isso permite que o thread instigador "esqueça" o delegado assíncrono, mas requer um pouco de trabalho extra no final do retorno de chamada:

static int Work(string s) { return s.Length; }

static void Done(IAsyncResult cookie)

{

var target = (Func<string, int>)cookie.AsyncState;

int result = target.EndInvoke(cookie);

Console.WriteLine("String length is: " + result);

}

Func<string, int> method = Work;

method.BeginInvoke("test", Done, method);

O argumento final para BeginInvoke é um objeto de estado do usuário que preenche a propriedade AsyncState de IAsyncResult. Pode conter qualquer coisa que você quiser; nesse caso, estamos usando-o para passar o delegado do método para o retorno de chamada de conclusão, para que possamos chamar EndInvoke nele.

**Thread.Abort**

Para parar um thread, você pode usar o método Thread.Abort. Esse método gera um ThreadAbortException no thread no qual ele é invocado, para iniciar o processo de encerramento do thread. Geralmente, a chamada a esse método termina o thread. No entanto, como esse método é executado por outro thread, pode acontecer a qualquer momento. Quando isso acontece, um ThreadAbortException é lançado no thread de destino. Isso pode deixar um estado corrompido e tornar seu aplicativo inutilizável. Por exemplo, chamar Thread.Abort pode impedir que construtores estáticos executem ou impeçam o lançamento de recursos não gerenciados.

Para encerrar a execução do thread, você geralmente usa o modelo de cancelamento cooperativo. Às vezes, não é possível parar um thread de cooperativa, pois ele executa código de terceiros não projetado para cancelamento cooperativo. O Thread.Abort método no .NET Framework pode ser usado para encerrar forçosamente um thread gerenciado. Quando você chama Abort, o Common Language Runtime gera um ThreadAbortException no thread de destino, que pode ser detectado pelo thread de destino.

static void TestMethod()

{

try

{

while (true)

{

Console.WriteLine("New thread running.");

Thread.Sleep(1000);

}

}

catch (ThreadAbortException abortException)

{

Console.WriteLine((string)abortException.ExceptionState);

}

}

Thread newThread = new Thread(new ThreadStart(TestMethod));

newThread.Start();

Thread.Sleep(1000);

// Abort newThread.

Console.WriteLine("Main aborting new thread.");

newThread.Abort("Information from Main.");

// Wait for the thread to terminate.

newThread.Join();

Console.WriteLine("New thread terminated - Main exiting.");

Após a anulação de um thread, ele não poderá ser reiniciado. Se Abort for chamado em um thread que não foi iniciado, o thread será anulado quando Start for chamado. Se Abort for chamado em um thread bloqueado ou estiver em suspensão, o thread será interrompido e anulado. Se Abort for chamado em um thread que foi suspenso, um ThreadStateException será gerado no thread que chamou Aborte AbortRequested será adicionado à propriedade ThreadState do thread que está sendo anulado. Uma ThreadAbortException não é lançada no thread suspenso até que Resume seja chamado. Se Abort for chamado em um thread gerenciado enquanto estiver executando código não gerenciado, um ThreadAbortException não será lançado até que o thread retorne para o código gerenciado.

O método Abort não causa a anulação imediata do thread, porque o thread de destino pode capturar o ThreadAbortException e executar valores arbitrários de código em um bloco finally. Você poderá chamar Thread.Join se precisar esperar até que o thread seja encerrado. Thread.Join é uma chamada de bloqueio que não retorna até que o thread realmente tenha parado de executar ou um intervalo de tempo limite opcional tenha transcorrido. O thread anulado poderia chamar o método ResetAbort ou executar o processamento não associado em um bloqueio finally, então se você não especificar um tempo limite, não será garantido o término da espera.

O Thread.Abort método não tem suporte no .NET Core. Se você precisar encerrar a execução de código de terceiros de modo forçado no .NET Core, execute-o no processo e uso separados Process.Kill . Uma maneira melhor de interromper um thread é usando uma variável compartilhada para que seu destino e seu thread de chamada possam acessar. O código abaixo mostra um exemplo.

private static bool stopped = false;

static void Main(string[] args)

{

Thread newThread = new Thread(new ThreadStart(TestMethod));

newThread.Start();

Thread.Sleep(1000);

//t.Start();

Console.WriteLine("Press any key to exit");

Console.ReadKey();

stopped = true;

newThread.Join();

}

static void TestMethod()

{

try

{

while (!stopped)

{

Console.WriteLine("New thread running.");

Thread.Sleep(1000);

}

}

catch (ThreadAbortException abortException)

{

Console.WriteLine((string)abortException.ExceptionState);

}

}

**TRABALHANDO COM TAREFAS**

Na maioria das vezes, você precisa executar algum trabalho de forma unitária e recuperar o resultado sem precisar se preocupar com a implementação subjacente no sistema operacional. Com o .NET 4, a Microsoft apresentou uma nova biblioteca, a TPL (Task Parallel Library), que é uma coleção de classes projetadas para abstrair os threads. Task é uma parte importante da Task Parallel Library, essa unidade de trabalho representa uma operação assíncrona e pode ser executada independentemente.

|  |  |  |
| --- | --- | --- |
| **Comparação** | **Threads** | **Tasks** |
| Namespace | System.Threading | System.Threading.Tasks |
| Definição | 1. É uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrentemente. 2. É algo mais próximo do concreto. É um dos muitos possíveis trabalhadores que executa essa tarefa. | 1. Representa uma unidade de trabalho que deverá ser realizada, é uma promessa, ou seja, uma Task<T> promete devolver um T em algum momento mais tarde. 2. É algo mais abstrato. É algo que você deseja fazer, está se dizendo que precisa de algo pronto em algum momento futuro. |
| Exemplo | Thread.Sleep() consome processamento para esperar um tempo | [Task.Delay()](https://pt.stackoverflow.com/q/86014/101) cria uma interrupção no processador (através do OS) para o código ser invocado. |
| Retorno | Não há mecanismo direto para retornar o resultado. | Pode retornar um resultado |
| Exceção | Se ocorrer uma exceção não é possível capturá-la na função Pai. | Uma Task filha pode propagar para a Task pai. |
| Cancelamento | Não é possível cancelar um vez enfileirada na ThreadPool. | Suporta o cancelamento de tarefas através do uso de tokens de cancelamento |
| Operação assíncrona | São usadas ​​para concluir operação assíncrona, quebrando o trabalho em pedaços e atribuindo-os as threads separados em segundo plano. | Representa uma operação assíncrona. Podemos usar as palavras chaves 'async' e 'await' para implementar facilmente operações assíncronas em uma Task ou Tasks encadeadas(pai/filho). |
| Custo | Uma thread quase sempre não é desejável, pois tem um alto custo, assim é mais fácil reutilizar uma thread existente do ThreadPool. | Nem todas as Tasks precisam de uma nova Thread. Se o sistema tiver várias Tasks, então ele faz uso do ThreadPool e, portanto, não tem a sobrecarga associada à criação de uma thread dedicada usando uma Thread. Reduz o tempo de comutação entre vários threads. |
| Thread | Ao iniciar um thread, esse thread compete com outros threads da CPU para executar. Às vezes, um thread é interrompido no meio de uma operação e deve aguardar sua vez de executar novamente para poder concluir essa operação. O thread interrompe todas as threads de segundo plano quando a de primeiro plano termina. | Oferece melhor controle programático para executar e aguardar uma tarefa. Permite encadear várias tarefas (relacionamento pai/filho) e pode executar cada tarefa uma após a outra usando ContinueWith().Task aguarda que todos os objetos de tarefas fornecidos concluam a execução. |

using System;

using System.Threading;

using System.Threading.Tasks;

Console.WriteLine("Iniciando uma Thread com delay de 5 s");

//Threads

Thread th = new Thread(() =>

{

Thread.Sleep(5000);

});

Console.WriteLine("Iniciando uma Task com delay de 5 s");

//Tasks

Task t = Task.Factory.StartNew(() =>

{

Thread.Sleep(5000);

});

Ao executar o código você vai notar que na execução da Thread vai esperar pelos 5 segundos até terminar. Já a execução da Task ocorre instantâneamente sem o delay de 5 segundos. O que acontece é que no caso da Task é criada uma thread em background para a qual a thread principal não vai aguardar a execução para concluir.

**Task**

Uma tarefa representa uma unidade de trabalho que deverá ser realizada. Esta unidade de trabalho pode rodar em uma thread separada e é também possível iniciar uma task de forma sincronizada a qual resulta em uma espera pela thread chamada. Com tarefas, você tem uma camada de abstração, mas também um bom controle sobre as threads relacionadas.

As tarefas permitem muito mais flexibilidade na organização do trabalho que você precisa fazer. Por exemplo, você pode definir continuar o trabalho, que deve ser feito depois que uma tarefa esteja completa. Isso pode diferenciar se uma tarefa foi executada com sucesso ou não. Você também pode organizar as tarefas em hierarquia onde uma tarefa pai pode criar novas tarefas filhas que pode criar dependências e assim o cancelamento da tarefa pai também cancela suas tarefas filhas. A tarefa não cria novos threads, em vez disso, gerencia com eficiência os threads do ThreadPool. As tarefas são executadas pelo TaskScheduler, que enfileira tarefas em threads.

**Trabalhando com o TaskScheduler**

O trabalho de enfileirar tarefas nos threads é realizado por um componente chamado agendador de tarefas, implementado pela classe TaskScheduler. Normalmente, você não trabalha diretamente com o agendador. Quando você inicia uma nova tarefa, se não estiver especificando nenhum agendador, ele usará um padrão. Há uma situação, no entanto, que você precisa usar o agendador ao usar tarefas e é quando você usa tarefas em um aplicativo Windows Forms ou WPF. Nestes tipo de aplicações, a interface do usuário poderá ser atualizada apenas pelo thread da interface do usuário (utilizando o método Invoke) senão congelará a tela para usuário, como abaixo:

static int Soma(object imageOne, object imageTwo, object imageThree)

{

return (int)imageOne + (int)imageTwo + (int)imageThree;

}

var tf = Task<int>.Factory;

// Load the three images asynchronously

var imageOne = tf.StartNew(() => { Thread.Sleep(100); return 10; });

var imageTwo = tf.StartNew(() => { Thread.Sleep(1000); return 50; });

var imageThree = tf.StartNew(() => { Thread.Sleep(50000); return 30; });

// When they’ve been loaded, blend them

var blendedImage = tf.ContinueWhenAll(

new[] { imageOne, imageTwo, imageThree }, \_ =>

Soma(imageOne.Result, imageTwo.Result, imageThree.Result));

Task.WhenAll(imageOne, imageTwo, imageThree);

label1.Text = (int)imageOne + (int)imageTwo + (int)imageThree;

Para conseguir atualizar a inteface do usuário, você precisa chamar uma das sobrecargas StartNew ou ContinueWith que usa um parâmetro TaskScheduler e passa TaskScheduler.FromCurrentSynchronizationContext() como o valor para esse parâmetro.

var ui = TaskScheduler.FromCurrentSynchronizationContext();

// When they’ve been loaded, blend them

var blendedImage = tf.ContinueWhenAll(

new[] { imageOne, imageTwo, imageThree }, \_ =>

Soma(imageOne.Result, imageTwo.Result, imageThree.Result));

// When we’re done blending, display the blended image

blendedImage.ContinueWith(\_ =>

{

label1.Text = blendedImage.Result.ToString();

}, ui);

Um exemplo um pouco mais complexo que utiliza o padrão:

* Task.Factory.StartNew(() => Método/Action, CancellationToken.None, TaskCreationOptions.None, TaskScheduler.FromCurrentSynchronizationContext());

TaskScheduler uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();

Task.Factory.StartNew(() =>

{

for (int i = 0; i < 10; i++)

{

Task.Factory.StartNew(() =>

{

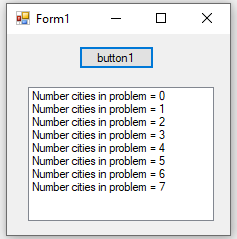
listBox1.Items.Add("Number cities in problem = " + i.ToString());

}, CancellationToken.None, TaskCreationOptions.None, uiScheduler);

System.Threading.Thread.Sleep(1000);

}

}, CancellationToken.None, TaskCreationOptions.None, TaskScheduler.Default);



Criando a tarefa dessa maneira, ela será executada pelo thread da interface do usuário assim que o thread da interface do usuário puder processá-lo.

**Iniciando Tasks**

Para criar uma tarefa que não retorna um valor, usamos uma classe Task do namespace System.Threading.Tasks. Ele contém alguns métodos e propriedades importantes que são úteis para gerenciar a operação da tarefa:

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| CurrentId | Retorna o ID da tarefa atualmente em execução. |
| Exception | Retorna os dados não manipulados AggregateException, se houver, que causou o término da execução da tarefa. |
| Factory | Retorna um objeto de fábrica que pode ser usado para criar e configurar uma nova tarefa. |
| ID | Obtém o ID de uma instância de tarefa específica. |
| IsCanceled | Obtém um valor booleano para determinar se uma tarefa é cancelada. |
| IsCompleted | Obtém um valor booleano para determinar se uma tarefa foi concluída |
| IsFaulted | Obtém se a tarefa é concluída devido a uma exceção não tratada. |
| Status | Obtém o status da tarefa atual |
| Result | Obtém o valor retornado pela operação assíncrona representada por esta tarefa |

|  |  |
| --- | --- |
| **Métodos** | **Descrição** |
| ContinueWith() | Cria uma nova tarefa que será iniciada de forma assíncrona quando a tarefa atual for concluída. |
| Delay() | Este método estático cria uma tarefa que é marcada como concluída após o atraso especificado. |
| Run() | Esse método estático adiciona uma solicitação de trabalho no ThreadPool e retorna uma tarefa |
| Start() | Inicia a tarefa representada por esta instância. |
| Wait() | Aguarda a conclusão da tarefa representada por esta instância especificada para concluir sua execução. |
| WaitAll() | Este método estático aguarda a conclusão de todas as tarefas enviadas como parâmetros. |
| WaitAny() | Este método estático aguarda a conclusão de qualquer uma das tarefas enviadas como parâmetros. |
| WhenAll() | Este método estático cria uma tarefa que é marcada como concluída quando todas as tarefas enviadas como parâmetros são concluídas. |
| WhenAny() | Este método estático cria uma tarefa que é marcada como concluída quando qualquer uma das tarefas enviadas como parâmetros é concluída. |

A propriedade estática Factory é do tipo TaskFactory e é usada para criar novas tarefas. A tabela abaico descreve alguns dos métodos mais comuns.

|  |  |
| --- | --- |
| **Métodos** | **Descrição** |
| ContinueWhenAll | Cria uma tarefa que inicia quando todas as tarefas enviadas como parâmetros são concluídas. |
| ContinueWhenAny | Cria uma tarefa que inicia quando qualquer uma das tarefas enviadas como parâmetros é concluída. |
| FromAsync | Vários métodos sobrecarregados usados para trazer o código APM antigo para o novo modelo TAP, envolvendo uma tarefa em torno da chamada assíncrona |
| StartNew | Vários métodos sobrecarregados usados para criar uma tarefa e iniciá-la. |

A tarefa é uma parte importante da programação assíncrona e é executada em um thread ThreadPool. Geralmente, uma expressão lambda é usada para especificar o trabalho que a Tarefa deve executar.

Existem duas classes de tarefas:

* **Task:** é usado quando você executa um método como uma tarefa e não possui valor de retorno
* **Task<TResult>:** é usado quando você executa uma função como uma tarefa com valor de retorno.

Você pode criar uma tarefa de várias maneiras:

* **Pelo método estático TaskFactory.StartNew:** esse método cria e inicia as tarefas. Por razões de desempenho, o método StartNew() é o mecanismo recomendado pois iniviam a tarefa imediatamente
* **Pelo método estático Task.Run:** esses método cria e inicia as tarefas. Este é um invólucro simplificado para TaskFactory.StartNew, mas com mais eficiência.
* **Pelo método de instância Task:** esse método é indicado em situações em que a criação e programação da tarefa devem ser separadas. Quando o objeto Task é instanciado é dado o status Created e a tarefa não é executada imediatamente. Para iniciar a tarefa você deve chamar o método Start.
* **Por métodos de continuação:** esses são Task.ContinueWith, Task.WhenAll, Task.WhenAny, TaskFactory.ContinueWhenAll, TaskFactory.ContinueWhenAny.

**MÉTODO TASKFACTORY.STARTNEW**

O TaskFactory.StartNew oferece uma grande flexibilidade. Ao criar uma nova tarefa, você precisa especificar pelo menos o método ou a função que deseja executar como tarefa. Além disso, você pode especificar opções para criar a tarefa, um token de cancelamento e um planejador que enfileire tarefas em threads. (Os agendadores são discutidos na próxima seção e o cancelamento será discutido mais adiante neste capítulo na seção "Trabalhando com cancelamentos".)

A enumeração TaskCreationOptions descreve as opções para criar tarefas. Essa enumeração é decorada com o FlagsAttribute, o que significa que essas opções podem ser combinadas. A tabela abaixo descreve as opções

|  |  |
| --- | --- |
| **Nome do Membro** | **Descrição** |
| None | Comportamento padrão. |
| PreferFairness | As tarefas devem ser agendadas de maneira justa. O resultado pretendido é FIFO , ou seja, tarefas agendadas mais cedo terão uma chance de serem executadas mais cedo, e as tarefas agendadas mais tarde serão executadas mais tarde. |
| LongRunning | Isso é usado para especificar que a tarefa levará muito tempo para ser concluída. Esta é apenas uma dica e o resultado será uma assinatura, que permite que o planejador crie mais threads para executar as tarefas do que o número disponível de threads de hardware. |
| AttachedToParent | A tarefa recém-criada é anexada à tarefa pai na hierarquia. |
| DenyChildAttach | Especifica que nenhuma tarefa filho pode ser anexada à tarefa atual. Se você tentar anexar uma tarefa filho a essa tarefa recém-criada, um InvalidOperationException será lançada. |
| HideScheduler | Especifica que o planejador atual não deve ser usado ao criar novas tarefas a partir desta tarefa recém-criada. Essas novas tarefas devem usar Default como o planejador atual quando elas são criadas |

Por razões de desempenho, o método StartNew da classe TaskFactory deve ser o mecanismo preferido para criação e programação de tarefas pois, ao utilizar o método StartNew(), a tarefa é iniciada imediatamente. Para iniciar uma tarefa utilizando o método StartNew você pode:

* **Usar TaskFactory**

static void FazerAlgo()

{

Console.WriteLine("executando uma tarefa => FazerAlgo()");

}

TaskFactory tf = new TaskFactory();

Task t = tf.StartNew(FazerAlgo);

* **Usar classe Task**

Task t1 = Task.Factory.StartNew(() => FazerAlgo());

Task t2 = Task.Factory.StartNew(FazerAlgo);

Abaixo um exemplo mais prático que reutiliza uma tarefa usando Task.Factory.StartNew

static void myMethod()

{

Console.WriteLine("Hello From My Task");

for (int i = 0; i < 10; i++)

{

Console.Write("{0} ", i);

}

Console.WriteLine();

Console.WriteLine("Bye From My Task");

}

//initialize and Start mytask and assign

//a unit of work in the body of lambda exp

Task mytask = Task.Factory.StartNew(new Action(myMethod));

mytask.Wait(); //Wait until mytask finish its job

Console.WriteLine("Hello From Main Thread");

Use Task<T>.Factory.StartNew para retornar um valor de um método Task:

static int myMethodreturnvalue()

{

Console.WriteLine("Hello from myTask<int>");

Thread.Sleep(1000);

return 10;

}

Task<int> myTaskreturnvalue = Task<int>.Factory.StartNew(myMethodreturnvalue);

Console.WriteLine("Hello from Main Thread returns value");

//Wait the main thread until myTaskreturnvalue is finished

//and returns the value from myTaskreturnvalue operation (myMethod)

int i = myTaskreturnvalue.Result;

Console.WriteLine("myTask has a return value = {0}", i);

Console.WriteLine("Bye From Main Thread returns value");

Crie uma Task <T> e inicie-a imediatamente chamando o método StartNew. No .NET 4.0, é preferível usar a Task <T> .Factory.StartNew para criar e iniciar uma tarefa porque economiza custos de desempenho, enquanto que a Task<T> (...). Start () consome mais custos de desempenho para criar e iniciar iniciando uma tarefa.

**MÉTODO ESTÁTICO TASK.RUN**

Com a classe de Task, ao invés de invocar o método Start(), você pode invocar o método **RunSynchronously**(). Desta forma, a tarefa é iniciada também, mas ela está sendo executada na thread atual do chamador, o chamador precisa esperar até que a tarefa termine. Por padrão, a tarefa é executada de forma assíncrona.

O Task.Run() retorna e executa uma tarefa atribuindo uma unidade de trabalho na forma de um método ("myMethod"). No .NET 4.5, é preferível usar o Task.Run porque gerencia o Task com mais eficiência do que o Task.Factory.StartNew. Isso de forma alguma obsoleta o Task.Factory.StartNew, mas deve ser simplesmente considerado uma maneira rápida de usar o Task.Factory.StartNew sem a necessidade de especificar vários parâmetros. De fato, o Task.Run é realmente implementado em termos da mesma lógica usada para Task.Factory.StartNew, apenas transmitindo alguns parâmetros padrão. Quando você passa uma ação para Task.Run:

Task.Run(someAction);

isso é exatamente equivalente a:

Task.Factory.StartNew(someAction, CancellationToken.None, TaskCreationOptions.DenyChildAttach, TaskScheduler.Default);

//initialize and Run mytask and assign

//a unit of work in form of 'myMethod()'

Task mytask = Task.Run(new Action(myMethod));

mytask.Wait(); //Wait until mytask finish its job

Task mylamnbdaTask = Task.Run(() =>

{

Console.WriteLine("Hello From My Task");

for (int i = 0; i < 10; i++)

{

Console.Write("{0} ", i);

}

Console.WriteLine();

Console.WriteLine("Bye From My Task");

});

mylamnbdaTask.Wait(); //Wait until mytask finish its job

Console.WriteLine("Hello From Main Thread");

Console.ReadKey();

A expressão lambda é mais comumente usada para atribuir um método anônimo a Task.Run(). O myTask executará o método anônimo em uma tarefa separada.

Use Task.Run <int> para retornar um valor do método Task:

Task<int> myTaskreturnvalue = Task.Run<int>(new Func<int>(myMethodreturnvalue));

Console.WriteLine("Hello from Main Thread returns value");

//Wait for the main thread until myTask is finished

//and return the value from myTask operation (myMethod)

int valor = myTaskreturnvalue.Result;

Console.WriteLine("myTask has a return value = {0}", valor);

Console.WriteLine("Bye From Main Thread returns value");

O Task.Run<int>() pega um delegado Func<int> para fazer referência a um método que retorna um valor inteiro. Este método é executado por uma tarefa e um valor é retornado usando a propriedade Result.

Use a expressão lambda para retornar um valor do método de uma tarefa.

Task<int> mylamnbdaTaskreturnvalue = Task.Run<int>(() =>

{

Console.WriteLine("Hello from myTask<int>");

Thread.Sleep(1000);

return 10;

});

Se utilizar a palavra-chave Var não é necessário declarar o tipo retornado em Task.Run. A palavra-chave Var detecta o tipo de tarefa <T> procurando no valor de retorno da expressão lambda.

var myTask = Task.Run(() =>

{

Console.WriteLine("Hello from myTask<int>");

Thread.Sleep(1000);

return 10;

});

A expressão Lambda pode ser usada para definir uma unidade de trabalho para a Task<int>. E, dentro da expressão lambda, seu valor de retorno deve corresponder ao tipo de Task<T>.

**Tarefa aninhada (Nested Task)**

Uma tarefa aninhada é apenas uma instância de tarefa criada no delegado do usuário de outra tarefa. Uma tarefa filho é uma tarefa aninhada criada com a opção AttachedToParent. Uma tarefa pode criar qualquer número de tarefas filho e/ou aninhadas, limitadas apenas pelos recursos do sistema. O exemplo a seguir mostra uma tarefa pai que cria uma tarefa aninhada simples. Toda tarefa aninhada é por padrão uma tarefa filho desanexada. Ele roda independentemente de seu pai

Task outer = Task.Run(() =>

{

Console.WriteLine("Hi I'm outer task ");

Task inner = Task.Run(() =>

{

Console.WriteLine("Hi I'm inner task");

Thread.Sleep(2000);

Console.WriteLine("By from inner task");

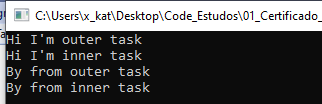
});

Thread.Sleep(500);

Console.WriteLine("By from outer task");

});

outer.Wait();



Podemos criar uma tarefa interna, tanto quanto desejamos. Mas as tarefas internas e externas serão executadas independentemente uma da outra. Quando uma tarefa externa concluir sua execução, ela sairá e será sincronizada com o thread principal.

**Tarefa filho anexada ao pai**

Uma tarefa filho aninhada pode se conectar ao pai usando a opção AttachedToParent. A tarefa pai não pode terminar sua execução até que todas as tarefas filho anexadas concluam sua execução

Task outer = new Task(() =>

{

Console.WriteLine("ToParent: Hi I'm outer task ");

//AttachedToParent only available with new Task()

Task inner = new Task(() =>

{

Console.WriteLine("ToParent: HI I'm inner task");

Thread.Sleep(2000);

Console.WriteLine("ToParent: By from inner task");

}, TaskCreationOptions.AttachedToParent);

inner.Start();

Thread.Sleep(500);

Console.WriteLine("ToParent: By from outer task");

});

outer.Start();

outer.Wait();

É importante não usar "Task.Run ()" ao criar uma tarefa filho que depende do pai. No trecho de código acima, uma nova tarefa aninhada foi criada e anexada ao pai usando a propriedade "AttachedToParent" como o segundo argumento de "new Task()". Um exemplo de tarefas aninhadas utilizando TaskFactory:

Task<Int32[]> parent = Task.Run(() =>

{

var results = new Int32[3];

TaskFactory tf = new TaskFactory(TaskCreationOptions.AttachedToParent,

TaskContinuationOptions.ExecuteSynchronously);

tf.StartNew(() => results[0] = 0);

tf.StartNew(() => results[1] = 1);

tf.StartNew(() => results[2] = 2);

return results;

});

var finalTask = parent.ContinueWith(

parentTask => {

foreach (int i in parentTask.Result)

Console.WriteLine(i);

});

finalTask.Wait();

**MÉTODO DE INSTÂNCIA TASK**

Quando o objeto Task é instanciado, a tarefa **não será executada imediatamente**. Em vez disso, a ela é dado o status **Created**. A tarefa é, então iniciada pela chamada do método **Start**() da classe Task, sendo indicado em situações em que a criação e programação da tarefa devem ser separadas.

O construtor Task lhe dá mais flexibilidade, pois ao iniciar uma tarefa, uma instância da classe Task pode ser criada e o código que deve ser executado pode ser atribuído com uma Action ou delegate Action<object> tanto sem parâmetro como com um parâmetro object. Isto é semelhante ao que você viu na classe Thread.

**Tarefa mytask = new Task (actionMethod);**

Onde:

1. **actionMethod**: é um método que possui um tipo de retorno nulo e não aceita parâmetros de entrada; em outras palavras, há um delegado "Action" no parâmetro do construtor Task.
2. **Task**: possui um total de 8 construtores sobrecarregados, mas geralmente trabalhamos com o primeiro construtor sobrecarregado que possui um delegado "Action" em seu parâmetro de entrada.

private static void myMethod()

{

Console.WriteLine("Hello From My Task");

for (int i = 0; i < 10; i++)

{

Console.Write("{0} ", i);

}

Console.WriteLine();

Console.WriteLine("Bye From My Task");

}

//initialize mytask and assign a unit of work in form of 'myMethod()'

Task myTask = new Task(myMethod);

myTask.Start();// Start the execution of mytask

myTask.Wait(); //Wait until mytask finish its job

Console.WriteLine("Bye From Main Thread");

Sabemos que o Task executa nos threads de segundo plano de um pool de threads. Portanto, é importante escrever um método "Wait()", caso contrário, o programa será encerrado assim que o Main Thread terminar sua execução.

**Task<Result>**

A Task<Result> é usada com operações assíncronas que retornam um valor. A classe Task<Result> é encontrada no namespace System.Threading.Task e herda da classe Task.

static int myMethodreturnvalue()

{

Console.WriteLine("Hello from myTask<int>");

Thread.Sleep(1000);

return 10;

}

Task<int> myTaskreturnvalue = new Task<int>(myMethodreturnvalue);

myTaskreturnvalue.Start(); //start myTask

Console.WriteLine("Hello from Main Thread returns value");

//Wait the main thread until myTaskreturnvalue is finished

//and returns the value from myTaskreturnvalue operation (myMethod)

int i = myTaskreturnvalue.Result;

Console.WriteLine("myTask has a return value = {0}", i);

A Task<int> informa à operação da tarefa para retornar um valor inteiro. myTask.Result; é uma propriedade que retorna um valor quando a tarefa é concluída e bloqueia a execução de um thread de chamada (nesse caso, seu thread principal) até que a tarefa conclua sua execução.

**TaskStatus**

Utilize o enumerador TaskStatus(tarefa.Status == TaskStatus.Created) para verificar o Status da tarefa.

|  |  |  |
| --- | --- | --- |
| **Situação** | **Enum** | **Descrição** |
| Canceled | 6 | A tarefa reconheceu o cancelamento lançando uma OperationCanceledException com seu próprio CancellationToken enquanto o token estava no estado sinalizado ou o CancellationToken da tarefa já estava sinalizado antes que a tarefa começasse a ser executada. |
| Created | 0 | A tarefa foi inicializada, mas ainda não foi agendada. |
| Faulted | 7 | A tarefa foi concluída devido a uma exceção sem tratamento. |
| RanToCompletion | 5 | A execução da tarefa foi concluída com êxito. |
| Running | 3 | A tarefa está em execução, mas ainda não foi concluída. |
| WaitingForActivation | 1 | A tarefa está aguardando para ser ativada e agendada internamente pela infraestrutura do .NET Framework. |
| WaitingForChildrenToComplete | 4 | A tarefa terminou de ser executada e está aguardando implicitamente a conclusão de tarefas filho anexadas. |
| WaitingToRun | 2 | A tarefa foi agendada para execução, mas ainda não começou a ser executada. |

Task t3 = new Task(myMethod);

t3.Start();

Task<int>[] tarefas = new Task<int>[2];

tarefas[0] = new Task<int>(() => { return 34; });

var situacao = tarefas[0].Status;

Console.WriteLine("Status == " + situacao); //Status == Created

tarefas[0].Start();

situacao = tarefas[0].Status;

Console.WriteLine("Status == " + situacao); //Status == WaitingToRun

Chamar Task.Start em uma situação que não seja Created pode levar a várias exceções, como InvalidOperationException: 'Start não pode ser chamado em uma tarefa estilo promessa.':

static async Task FazerAlgo()

{

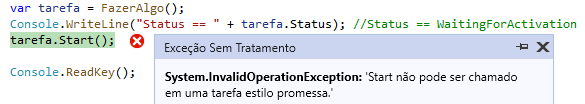
await Task.Delay(TimeSpan.FromSeconds(3));

}

var tarefa = FazerAlgo();

Console.WriteLine("Status == " + tarefa.Status); //Status == WaitingForActivation

tarefa.Start();



Ou como InvalidOperationException: 'Start não pode ser chamado em uma tarefa já concluída.:

static async Task TarefaConcluida()

{

Console.WriteLine("Tarefa realizada");

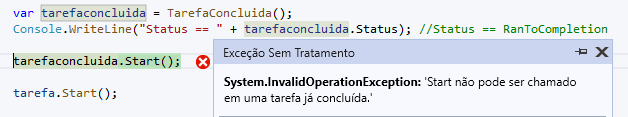
}

var tarefaconcluida = TarefaConcluida();

Console.WriteLine("Status == " + tarefaconcluida.Status); //Status == RanToCompletion

tarefaconcluida.Start();

TaskStatus == RanToCompletion >>**System.InvalidOperationException:** 'Start não pode ser chamado em uma tarefa já concluída.'



As tarefas são executadas de forma assíncrona em um thread do conjunto de threads. O conjunto de threads contém threads em segundo plano; portanto, quando a tarefa está em execução, o thread principal pode encerrar o aplicativo antes que a tarefa seja concluída. Para sincronizar a execução do thread principal e as tarefas assíncronas, usamos o método Wait.

**Aguarde uma ou mais tarefas**

O método Wait bloqueia a execução de um thread de chamada até que a execução de uma tarefa especificada seja concluída. A seguir, são apresentados métodos importantes de espera que ajudam a sincronizar um thread principal com as Tarefas.

|  |  |
| --- | --- |
| **Metodos** | **Descrição** |
| Wait() | Bloqueia o thread de chamada até que a tarefa especificada conclua sua execução. |
| Task myTask = Task.Run(() => { Thread.Sleep(1000);}); //1 Sec  myTask.Wait(); | |
| Wait  (milissegundos) | Bloqueia a execução de um thread de chamada até a tarefa especificada terminar ou um intervalo de tempo limite decorrido. |
| Task myTask = Task.Run(() => { Thread.Sleep(2000);}); //2 Sec  Task myTask2 = Task.Run(() => { Thread.Sleep(500);}); //1/2 Sec  myTask.Wait(1000);// wait for 1 sec  myTask2.Wait(1000);// wait for 1 sec | |
| WaitAll() | Bloqueia a execução de um thread de chamada até que todas as tarefas especificadas concluam sua execução. Todos os objetos de tarefa devem ser referenciados em uma única matriz. |
| Task myTask1 = Task.Run(() => { Thread.Sleep(100); }); //1/10 Sec  Task myTask2 = Task.Run(() => { Thread.Sleep(500); }); //1/2 Sec  Task[] allTasks = { tsk1, tsk2 };  Task.WaitAll(allTasks); | |
| WaitAll  (milissegundos) | Bloqueia a execução de um thread de chamada até que todas as tarefas especificadas terminem ou que um intervalo de tempo limite termine. |
| Task myTask1 = Task.Run(() => { Thread.Sleep(500); }); //1/2 Sec  Task myTask2 = Task.Run(() => { Thread.Sleep(2000); }); //2 Sec  Task[] allTasks = { tsk1, tsk2 };  Task.WaitAll(allTasks, 1200); | |
| WaitAny() | Bloqueia a execução de um thread de chamada até que a qualquer tarefa de uma coleção de tarefas conclua sua execução. |
| Task myTask1 = Task.Run(() => { Thread.Sleep(1000); }); //1 Sec  Task myTask2 = Task.Run(() => { Thread.Sleep(500); }); //1/2 Sec  Task[] allTasks = { tsk1, tsk2 };  Task.WaitAny(allTasks); | |
| WaitAny (milissegundos) | Bloqueia a execução de um thread de chamada até que qualquer tarefa de uma coleção de tarefas seja concluída ou que um intervalo de tempo limite termine. |
| Task myTask1 = Task.Run(() => { Thread.Sleep(500); }); //1/w Sec  Task myTask2 = Task.Run(() => { Thread.Sleep(2000); }); //2 Sec  Task[] allTasks = { tsk1, tsk2 };  Task.WaitAny(allTasks, 1200); | |

É comum para uma operação assíncrona, na conclusão, invocar uma segunda operação e passar os dados para ela. Na biblioteca Task.Parallel, a mesma funcionalidade é fornecida por tarefas de continuação. Uma tarefa de continuação é uma tarefa assíncrona que é invocada por outra tarefa (conhecido como a antecedente), quando ela termina.

**MÉTODOS DE CONTINUAÇÃO**

Em algumas situações, você não pode transformar tudo em tarefas sem interromper seu aplicativo. Você precisa cuidar das dependências impostas pelo seu algoritmo. Se você tiver dependências entre tarefas, como não pode iniciar a etapa 3 antes que as etapas 1 e 2 sejam concluídas, poderá usar alguns dos mecanismos de continuação disponíveis no TPL.

A seguir, estão quatro cenários principais:

* **Parallel.Invoke:** Tarefas sendo executadas independentes um do outro.
* **Task.ContinueWith:** Uma tarefa só pode ser executada após o término de outra tarefa.
* **Task.Factory.ContinueWhenAll:** Uma tarefa é executada somente após o término de várias outras tarefas.
* **Task.Factory.ContinueWhenAny:** Uma tarefa é executada quando qualquer tarefa, de várias outras tarefas, terminar.

**Parallel.Invoke**

Conforme discutido, as tarefas são abstrações que representam operações assíncronas executadas por threads. Embora sejam mais leves que os threads, às vezes você só precisa de uma abstração melhor para realizar esse tipo de trabalho multitarefa. É por isso que a Microsoft criou a classe Parallel. Esta classe faz parte do namespace System.Threading.Tasks. Esta classe possui três métodos estáticos, conforme descrito na tabela abaixo.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| For | Semelhante ao loop for, mas as iterações podem ser executadas em paralelo. Existem 12 sobrecargas para esse método, algumas delas aceitando um parâmetro ParallelOptions, outras usando ParallelLoopState para controlar o loop. |
| ForEach | Semelhante a um loop foreach, mas as iterações podem ser executadas em paralelo. Existem 20 sobrecargas para esse método, algumas delas aceitando um parâmetro ParallelOptions, outras usando ParallelLoopState para controlar o loop. |
| Invoke | Este método tentará executar as ações fornecidas em paralelo. Existem duas sobrecargas para esse método, ambas aceitando uma matriz de delegados Actions como execute. Uma das sobrecargas aceita um parâmetro ParallelOptions |

Como você pode ver, os três métodos mencionam a possibilidade de executar em paralelo, mas eles não garantem isso. O paralelismo envolve pegar uma determinada tarefa e dividi-la em um conjunto de tarefas relacionadas que podem ser executadas simultaneamente. Isso também significa que você não deve analisar seu código para substituir todos os seus loops por loops paralelos. Você deve usar a classe Parallel apenas quando seu código não precisar ser executado sequencialmente.

O aumento do desempenho com o processamento paralelo ocorre apenas quando você tem muito trabalho a ser executado que pode ser executado em paralelo. Para conjuntos de trabalho menores ou para trabalhos que precisam sincronizar o acesso a recursos, o uso da classe Parallel pode prejudicar o desempenho. A melhor maneira de saber se isso funcionará na sua situação é medir os resultados.

O resultado da execução do dessa implementação abaixo é imprevisível, o que significa que os métodos podem ser executados em qualquer ordem, mas, considerando a natureza independente das etapas, não deve importar.

static void Step1() { Thread.Sleep(1000); Console.WriteLine("Step1"); }

static void Step2() { Thread.Sleep(200); Console.WriteLine("Step2"); }

static void Step3() { Thread.Sleep(800); Console.WriteLine("Step3"); }

Step1(); Step2(); Step3(); //Step1 Step2 Step3

Parallel.Invoke(Step1, Step2, Step3); //Step2 Step3 Step1

Para executar o Parallel.Invoke com passagem de parâmetros e retorno de função siga o padrão abaixo:

private static string UpdateLabel(string text)

{

Console.WriteLine(text); // Stop, Walk

return "Voltei do Invoke";

}

string resultado ="";

Parallel.Invoke(() => { resultado = UpdateLabel("Stop, Walk"); });

Console.WriteLine(resultado); // Voltei do Invoke

**Parallel.For**

Veja agora um exemplo de como pode ser implementado com o Parallel.For:

static double DoIntensiveCalculations()

{

// We are simulating intensive calculations

// by doing nonsens divisions

double result = 100000000d;

var maxValue = Int32.MaxValue;

for (int i = 1; i < maxValue; i++)

{

result /= i;

}

return result + 10d;

}

static void RunParallelFor()

{

double result = 0d;

// Here we call same method several times in parallel.

Parallel.For(0, NUMBER\_OF\_ITERATIONS, i => {

result += DoIntensiveCalculations();

});

// Print the result

Console.WriteLine("The result is {0}", result);

}

Como você deve ter adivinhado, se você executar o código anterior, obtém um resultado incorreto pelo mesmo motivo que fez antes: condições da corrida. Para resolver o problema, você deve cuidar disso usando resultados provisórios (interim results). Você pode usar uma sobrecarga do método Parallel.For para resolver esse problema:

public static ParallelLoopResult For<TLocal>(int fromInclusive, int toExclusive, Func<TLocal> localInit, Func<int, ParallelLoopState, TLocal, TLocal> body, Action<TLocal> localFinally)

static void RunParallelForCorrected()

{

double result = 0d;

// Here we call same method several times.

//for (int i = 0; i < NUMBER\_OF\_ITERATIONS; i++)

Parallel.For(0, NUMBER\_OF\_ITERATIONS,

// Interim resul = 0d

() => 0d,

// result += DoIntensiveCalculations();

(i, state, interimResult) => interimResult + DoIntensiveCalculations(),

// Final step after the calculations

// we add the result to the final result

(lastInterimResult) => result += lastInterimResult

);

// Print the result

Console.WriteLine("The result is {0}", result);

}

**Parallel.Foreach**

Parallel.Foreach é usado para iterar um loop foreach em vários threads e processadores. Na maioria dos casos, o loop Parallel.Foreach é muito mais rápido que um loop foreach normal

int[] data = { 1, 2, 3, 4, 5 };

Parallel.ForEach<int>(data, (d) =>

{

Console.WriteLine(d);

});

Tente resistir ao desejo de transformar todos os seus loops foreach e foreach em seus equivalentes paralelos. Se você fizer isso, corre o risco de quebrar seu aplicativo. Como você viu no exemplo, foi fácil fazer exatamente isso. Se você sabe que as iterações são completamente independentes umas das outras e pode evitar as condições da corrida, então, por todos os meios, faça isso. Mas são grandes as chances de que nem todos os seus loops sejam tão simples, portanto, sempre é recomendável um pouco de análise e teste.

**ParallelLoopResult**

ParallelLoopState é usado como um parâmetro de entrada para alguns dos métodos For e ForEach. Possui dois métodos, Stop e Break, que você pode usar para interromper prematuramente a execução de um loop. Se você usa o Break em um método For, está instruindo o loop a parar de executar todas as iterações com um iterador maior que o da iteração atual.

ParallelLoopResult loopparalelo = Parallel.

For(0, 1000, (int i, ParallelLoopState loopState) =>

{

if (i == 500)

{

Console.WriteLine("Breaking loop");

loopState.Break();

}

return;

});

Ao interromper o loop paralelo, a variável resultante tem um valor IsCompleted false e um LowestBreakIteration de 500. Quando você usa o método Stop, o LowestBreakIteration é nulo.

**Task.ContinueWith**

O método Task.ContinueWith é usado para criar cadeias de várias tarefas. Cada próxima tarefa em uma cadeia não será agendada para execução até que a tarefa atual seja concluída com êxito, com falha devido a uma exceção não tratada ou com saída antecipada devido ao cancelamento.

Podemos criar continuações usando o método Task.**ContinueWith** que tem um parâmetro do tipo Task, onde pode-se acessar informações sobre a tarefa de origem (antecedente).As continuações são relativamente fáceis de utilizar, e também são muito eficientes e flexíveis. Por exemplo, você pode:

* Passar dados da antecessora para a continuação;
* Especificar as condições precisas em que a continuação será invocada ou não invocada;
* Cancelar uma continuação antes que ela comece ou cooperativamente enquanto ela estiver em execução.
* Fornecer dicas sobre como a continuação deve ser agendada;
* Invocar várias continuações da mesma antecessora.
* Invocar uma continuação quando todas ou qualquer uma das diversas antecessoras for concluída.
* Encadear continuações uma após a outra a qualquer comprimento arbitrário;
* Usar uma continuação para manipular exceções lançadas pela antecessora;

O exemplo a seguir mostra o padrão básico, (por motivos de clareza, o tratamento de exceção é omitido).

Task tarefaFactory = Task.Factory.StartNew(() =>

{

Thread.Sleep(100);

Console.WriteLine("tsk1");

});

// A continuacao. Seu delegate toma a tarefa antecedente

// como um argumento e pode retornar um tipo diferente

Task continuacao = tarefaFactory.ContinueWith((antecedent) =>

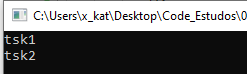
{

Thread.Sleep(500);

Console.WriteLine("tsk2");

});

continuacao.Wait();



O método tarefaFactory.ContinueWith executa e retorna uma nova tarefa quando o tarefaFactory concluir sua execução. Aqui o parâmetro antecedent do método ContinueWith é a referência de tarefaFactory. Esse antecedent pode ser utilizado no corpo de uma expressão lambda. Por exemplo, se tarefaFactory retornar um valor, usando antecedent o valor de retorno poderá ser usado no corpo de uma expressão lambda.

O método continuacao.Wait(); deve esperar a continuacao concluir sua execução, e sua execução será iniciada quando o tarefaFactory concluir sua execução. Portanto, continuacao.Wait() deve aguardar todas as tarefas que foram encadeadas com ele.

O Task<TResult> pode ser utilizável com continuação. Tal Task<TResult> retorna um valor para que uma nova tarefa em uma cadeia possa usá-lo.

// A tarefa antecedente. Pode tambem ser criada com Task.Factory.StartNew ou Task.Run

Task<DayOfWeek> tarefaA = new Task<DayOfWeek>(() => DateTime.Today.DayOfWeek);

Task<DayOfWeek> tarefaB = Task<DayOfWeek>.Run(() => DateTime.Today.AddDays(1).DayOfWeek);

// A continuacao. Seu delegate toma a tarefa antecedente

// como um argumento e pode retornar um tipo diferente

Task<string> continuacao = tarefaA.ContinueWith((antecedent) =>

{

return String.Format("Hoje é {0} ", antecedent.Result);

});

// Iniciar a antecedente

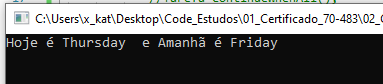
tarefaA.Start();

//Task.WhenAll(tarefaA, tarefaB);

Task.WaitAll(tarefaA, tarefaB);

// Usar o resultada da continuacao

Console.WriteLine(continuacao.Result + " e Amanhã é " + tarefaB.Result);



Utilizando somente ContinueWith a única garantia que você tem ao executar é que a continuacao é executada após a tarefaA. Nada pode ser dito sobre quando a tarefaB será executada. A última linha do código Task.WaitAll, garante a execução dos dois métodos tarefaA e tarefaB para coletar os resultados. Sem ele, o método Main retorna e o aplicativo pode não ter a chance de executar as tarefas. Você não precisa aguardar a tarefaA porque a continuacao é iniciada somente após a tarefaA terminar.

Invés do método Task.WaitAll poderia ser utilizao també o método Task.**WhenAll** que aguarda assincronamente múltiplas operações assíncronas que são representadas através de uma coleção de tarefas. A aplicação de WhenAll retorna uma única tarefa que não está completa até que cada tarefa na coleção seja concluída. As tarefas parecem ser executadas em paralelo, mas não são criados novas threads.

**Task.Factory.ContinueWhenAll**

Também é possível criar uma continuação multitarefa que será executada quando qualquer uma ou todas as tarefas de um array de tarefas tiverem sido completadas, como mostrado a seguir:

Task<int>[] tarefas = new Task<int>[2];

tarefas[0] = new Task<int>(() =>

{

// faz alguma coisa...

return 34;

});

tarefas[1] = new Task<int>(() =>

{

// faz alguma coisa...

return 8;

});

var continuation = Task.Factory.ContinueWhenAll(

tarefas,

(antecedents) =>

{

int resposta = tarefas[0].Result + tarefas[1].Result;

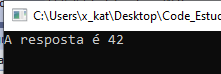
Console.WriteLine("A resposta é {0}", resposta);

});

tarefas[0].Start();

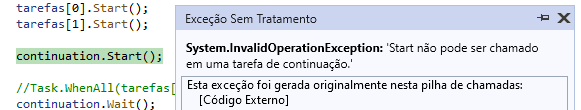
tarefas[1].Start();

continuation.Wait();



A chamada ContinueWhenAll tem como primeiro parâmetro uma matriz de tarefas e, como segundo parâmetro, um delegado para executar quando todas as tarefas terminarem. Ele retorna uma nova tarefa, que você pode usar para aguardar a conclusão de todas as tarefas. O delegado assume como parâmetro a matriz de tarefas que estava aguardando. A continuação é por si só uma tarefa e não bloqueia a thread na qual ela é iniciada. Use o método **Wait** para bloquear até a tarefa da continuação terminar.

Uma continuação é criada no estado WaitingForActivation e, portanto, só pode ser iniciada por sua tarefa antecedente. Chamar Task.Start em uma continuação no código do usuário levanta uma exceção System.InvalidOperationException.



**Task.Factory.ContinueWhenAny**

Ao chamar ContinueWhenAny, você cria uma tarefa que executa o delegado após a conclusão de qualquer tarefa da lista. O delegado toma como parâmetro a tarefa concluída. Se a tarefa concluída retornar algo, você poderá obter esse valor da propriedade previousTask.Result. Esse cenário é bastante comum quando você possui alguns serviços redundantes e se preocupa apenas com o valor recuperado pelo mais rápido.

static void Step1() { Thread.Sleep(1000); Console.WriteLine("Step1"); }

static void Step2() { Thread.Sleep(200); Console.WriteLine("Step2"); }

static void Step3() { Thread.Sleep(800); Console.WriteLine("Step3"); }

static void Metodos\_ContinueWhenAll()

{

Task step1Task = Task.Run(() => Step1());

Task step2Task = Task.Run(() => Step2());

Task step3Task = Task.Factory.ContinueWhenAny(

new Task[] { step1Task, step2Task },

(previousTask) => Step3());

step3Task.Wait();

}

**TaskContinuationOption**

TaskContinuationOption é uma enumeração usada para especificar quando uma tarefa em uma cadeia contínua é executada. A seguir, estão algumas das enumerações mais comuns para TaskContinuationOption:

|  |  |
| --- | --- |
| **Opções** | **Descrição** |
| OnlyOnFaulted | Especifica que a tarefa de continuação deve ser agendada apenas se seu antecedente lançou uma exceção não tratada. |
| NotOnFaulted | Especifica que a tarefa de continuação deve ser agendada se seu antecedente não lançar uma exceção não tratada. |
| OnlyOnCanceled | Especifica que a continuação deve ser agendada apenas se seu antecedente foi cancelado. Uma tarefa será cancelada se sua propriedade Task.Status após a conclusão for TaskStatus.Canceled. |
| NotOnCanceled | Especifica que a tarefa de continuação deve ser agendada se seu antecedente não tiver sido cancelado. |
| OnlyOnRanToCompletion | Especifica que a tarefa de continuação deve ser agendada se seu antecedente for concluído. |
| NotOnRanToCompletion | Especifica que a tarefa de continuação deve ser agendada se seu antecedente não for executado até a conclusão. |

Task<string> tsk1 = Task.Run(() =>

{

throw new Exception();

Console.WriteLine("tsk1 ran");

Thread.Sleep(100);

return "Ali";

});

Task tsk2 = tsk1.ContinueWith((t) =>

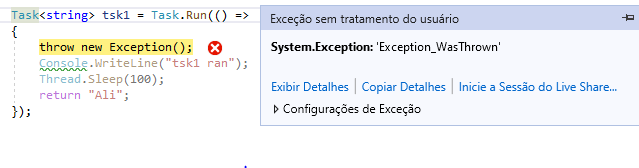
{

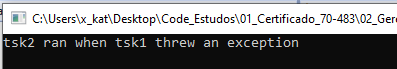
Console.WriteLine("tsk2 ran when tsk1 threw an exception");

}, TaskContinuationOptions.OnlyOnFaulted);

tsk2.Wait();

No exemplo acima, o segundo parâmetro do método tsk1.ContinueWith (TaskContinuationOptions) foi especificado com OnlyOnFaulted, que diz que o tsk2 só pode ser executado se o tsk1 lançou uma exceção sem tratamento, caso contrário, ignorará a execução do tsk2.





Da mesma forma, podemos especificar TaskContinuationOptions com outras enumerações, ou seja, OnlyOnCanceled, NotOnFaulted, etc. A tarefa retornada não será agendada para execução até que a tarefa atual seja concluída. Se os critérios de continuação especificados por meio do parâmetro continuationOptions não forem atendidos, a tarefa de continuação será cancelada em vez de agendada. Dessa forma, você pode adicionar diferentes métodos de continuação que serão executados quando ocorrer uma exceção, a tarefa for cancelada ou a tarefa for concluída com êxito. O código abaixo mostra como fazer isso.

Task<int> t = Task.Run(() =>

{

return 42;

});

t.ContinueWith((i) =>

{

Console.WriteLine("Canceled");

}, TaskContinuationOptions.OnlyOnCanceled);

t.ContinueWith((i) =>

{

Console.WriteLine("Faulted");

}, TaskContinuationOptions.OnlyOnFaulted);

var completedTask = t.ContinueWith((i) =>

{

Console.WriteLine("Completed");

}, TaskContinuationOptions.OnlyOnRanToCompletion);

**ASYNC AND AWAIT**

Na maioria das linguagens imperativas, incluindo as versões atuais do Visual Basic e do C#, a execução dos métodos (ou das funções, ou dos procedimentos) é contínua, isso significa que elas permitem que você expresse a sua lógica de programação como uma sequência de etapas separadas a serem executadas uma após a outra. Ou seja, uma vez que uma thread de controle comece a executar um determinado método, ela ficará ocupada com essa tarefa até que a execução do método seja concluída.

Às vezes, essa continuidade, esse sincronismo é um problema pois, não há nada que um método possa fazer para progredir, tudo o que ele pode fazer é esperar que algo aconteça: um download, o acesso a um arquivo, o cálculo executado em uma thread diferente, etc. Nesses casos, a thread fica totalmente ocupada sem fazer nada.

O termo geralmente usado nessas situações é: **thread bloqueado**; o **método** que causa isso é conhecido como **bloqueador**. Ao escrever programas de servidor, você não quer threads bloqueados. Você paga pelos threads bloqueados,esses threads poderiam servir a outras solicitações.

O comportamento síncrono pode deixar os usuários finais com uma má experiência e uma interface bloqueada/congelada sempre que o usuário tentar realizar alguma operação demorada. Nestes casos, uma abordagem assíncrona (threads) seria melhor. Uma chamada de método assíncrono (criação de uma thread) irá retornar imediatamente para que o programa possa realizar outras operações enquanto o método chamado conclui o seu trabalho em determinadas situações.

O comportamento do método assíncrono é diferente do síncrono, porque um método assíncrono é uma thread separada. Você cria a thread; a thread começa a executar, mas o controle é imediatamente retornado para a thread que a chamou, enquanto a outra thread continua a executar.Em geral, a programação assíncrona faz sentido em dois cenários:

1. Se você estiver criando uma aplicação com uma interface intensiva na qual a experiência do usuário é a principal preocupação. Neste caso, uma chamada assíncrona permite que a interface do usuário continue a responder e não fique congelada;
2. Se você tem um trabalho computacional complexo(muitos cálculos) ou muito demorado a fazer, e você tem que continuar interagindo com a interface do usuário do aplicativo enquanto espera a resposta de volta a partir da tarefa de longa duração;

Portanto, executar uma tarefa vinculada à CPU é diferente de uma tarefa vinculada à E/S. As tarefas ligadas à CPU sempre usam algum thread para executar seu trabalho. Uma tarefa associada a E/S assíncrona não usa um thread até que a E/S seja concluída.

Se você estiver criando um aplicativo cliente que precise permanecer responsivo enquanto as operações em segundo plano estiverem em execução, poderá usar a palavra-chave await para descarregar uma operação de longa execução para outro thread. Embora isso não melhore o desempenho, melhora a capacidade de resposta. A palavra-chave await também garante que o restante do seu método seja executado no thread correto da interface do usuário para que você possa atualizar a interface do usuário.

Criar um aplicativo escalável que use menos threads é outra história. Melhorar a escala do código significa alterar a implementação real do código. O xódigo abaixo mostra um exemplo disso.

public static string DebugInfo

{

get

{

ThreadPool.GetMaxThreads(out var maxThreads, out \_);

ThreadPool.GetAvailableThreads(out var threads, out \_);

var usedThreads = maxThreads - threads;

var mt = $"{usedThreads.ToString().PadLeft(4)}/{maxThreads.ToString().PadLeft(4)}";

return $"Threads {mt.PadRight(8)}";

}

}

public static Task SleepAsyncA(int millisecondsTimeout)

{

return Task.Run(() => { Console.WriteLine("SleepAsyncA " + DebugInfo); Thread.Sleep(millisecondsTimeout); });

}

public static Task SleepAsyncB(int millisecondsTimeout)

{

TaskCompletionSource<bool> tcs = null;

var t = new Timer(delegate { tcs.TrySetResult(true); }, null, -1, -1);

Console.WriteLine("SleepAsyncB " + DebugInfo);

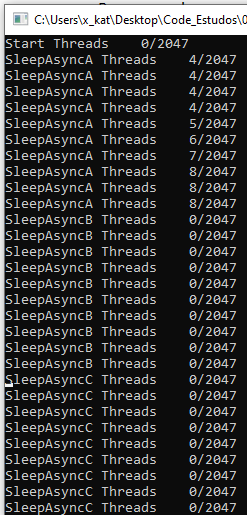
tcs = new TaskCompletionSource<bool>(t);

t.Change(millisecondsTimeout, -1);

return tcs.Task;

}

O método SleepAsyncA forçará um Thread do TPL ThreadPool a dormir pelo período de milissegundosTimeout, efetivamente tornando esse Thread inutilizável. É responsivo porque tem a aparência de assincronia mas não é escalável, porque vincula um recurso limitado (bloqueando um thread). O SleepAsyncB, usando um Timer, está atrasando um trecho de código para ser executado posteriormente até que milissegundosTimeout termine, em vez de bloquear uma thread. Isso não usa um thread, porque é uma funcionalidade nativa, fornecida pelo sistema operacional. O segundo método é responsivo e escalável, no entanto, enquanto SleepAsyncB é um bom exemplo de como usar construções de tarefas como TaskCompletionSource, ele pode ser substituído por uma única chamada para Task.Delay (millisecondsTimeout), como abaixo:



public static Task SleepAsyncC(int millisecondsTimeout)

{

Console.WriteLine("SleepAsyncC " + DebugInfo);

return Task.Delay(millisecondsTimeout);

}

Por exemplo, em escala, se você tivesse 100.000 chamadas pendentes no SleepAsyncA; ou você esgotaria o Threadpool e eles iniciariam na fila ou você teria 100.000 threads ativos, nenhum dos quais é muito bom para escalabilidade.O SleepAsyncB, por outro lado, usaria 0 threads enquanto as 100.000 chamadas estavam aguardando e não fazer nada é infinitamente mais escalável do que fazer alguma coisa.

var ms = 5000;

Console.WriteLine("Start " + DebugInfo);

var listA = Enumerable.Range(0, 10).Select(x => SleepAsyncA(ms));

Task.WaitAll(listA.ToArray());

var listB = Enumerable.Range(0, 10).Select(x => SleepAsyncB(ms));

Task.WaitAll(listB.ToArray());

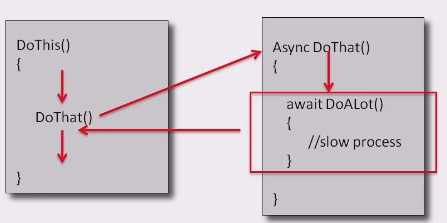
var listC = Enumerable.Range(0, 10).Select(x => SleepAsyncC(ms));

Task.WaitAll(listC.ToArray());

Ao usar as palavras-chaves async and await, lembre-se disso. Apenas agrupar todas as operações em uma tarefa e aguardá-las não fará com que seu aplicativo tenha um desempenho melhor. No entanto, poderia melhorar a capacidade de resposta, o que é muito importante nos aplicativos clientes.

A partir da versão 5.0 da linguagem C# o modificador async/wait e Async e Await no Visual Basic, oferecem uma maneira completamente diferente e fácil de fazer programação assíncrona. Ao utilizar essas duas palavras-chave, você pode usar os recursos do. NET Framework ou o Windows Runtime para criar um método assíncrono quase tão facilmente como você cria um método síncrono. Antes do C# 5.0, era necessário fazer uso de um BackgroundWorker para executar o thread da interface do usuário.

O método é executado de forma síncrona até que ele alcance a primeira expressão await, e neste ponto o método é suspenso até que a tarefa seja completada. Se um método que esta sendo modificado por uma palavra-chave async não contém uma expressão ou uma instrução await, o método é executado de forma síncrona. Um aviso do compilador o avisa sobre qualquer método assíncrono que não contiver um await porque essa situação pode indicar um erro.



static async Task DoStuff()

{

// If we comment out the await Task.Run instructions and

// everything happens synchronously...

await Task.Run(() =>

{

var t = CountToFifty();

});

// This code will not run until the CountToFifty call has completed

Console.WriteLine("Counting to 50 completed...");

}

private static async Task<string> CountToFifty()

{

int counter;

for (counter = 0; counter < 51; counter++)

Console.WriteLine("BG thread: " + counter);

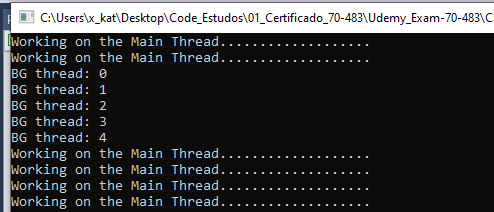
return "Counter = " + counter;

}

DoStuff();

for (int i = 0; i < 100; i++)

Console.WriteLine("Working on the Main Thread....");



Um método async (assíncrono) pode ter um tipo de retorno Task, Task(Of TResult) ou void. O método não pode declarar qualquer parâmetro ref ou out, embora ele pode chamar métodos que tenham esses parâmetros.

1. Você especifica Task<TResult> como o tipo de retorno de um método assíncrono se a declaração de retorno do método especifica um operando do tipo TResult.
2. Você usa Task se nenhum valor significativo é retornado quando o método for concluído.
3. O tipo de retorno void é usado principalmente para definir manipuladores de eventos, onde um tipo de retorno void é necessário.

Muitas das classes na .NET Framework Library que lidam com E/S foram modificadas adicionando a eles métodos assíncronos para dar suporte ao padrão async/await. Se você tem classes que lidam com E/S, você pode fazer o mesmo. Veja como você pode alterar um método síncrono existente para um método assíncrono. Aqui você tem o método ReadDataFromIO:

public static double ReadDataFromIO()

{

// We are simulating an I/O by putting the current thread to sleep.

Thread.Sleep(2000);

return 10d;

}

A variante assíncrona do método pode ser implementada de maneira simples:

public static Task<double> ReadDataFromIOAsync()

{

return Task.Run(new Func<double>(ReadDataFromIO));

}

Para tornar um método assíncrono, você deve retornar uma Task ou Task<TResult> e adicionar o sufixo Async ao nome do método. O sufixo existe para que os programadores que usam sua biblioteca saibam que o método é o equivalente assíncrono do seu método síncrono.

Quando um método é marcado com um modificador assíncrono, ele pode ter um dos três tipos de retorno a seguir: void, Task e Task<TResult>. Se seu método síncrono estava retornando void, você pode escolher entre void e Task. Se o método não for um manipulador de eventos (Controles de tela), a recomendação é retornar a Task. Ao retornar Task, você torna o método não apenas assíncrono, mas também awaitable. Se seu método síncrono estava retornando algo além de void, você deve alterar o tipo de retorno para Tarefa<TResult>. Como no exemplo acima, para que o método ReadDataFromIO retorne double o método variante assíncrona ReadDataFromIOAsync tem que ter como retorno Task<double>

|  |  |  |
| --- | --- | --- |
| lblResult.Content = await ReadDataFromIOAsync. | Equivalente a: | Task<double> task = ReadDataFromIOAsync();  lblResult.Content = task.Result; |

Em métodos assíncronos, você usa as palavras-chave e tipos para indicar o que você quer fazer, e o compilador faz o resto, incluindo manter o controle do que deve acontecer quando o controle retorna para um ponto await em um método suspenso. Existem algumas regras básicas que devem ser seguidas para o uso de async e await:

* A palavra-chave await não pode ser inserida em um dentro de uma instrução lock (que indica para uma aplicação que naquele momento alguma coisa será feita com o objeto que não pode ser interrompida);
* Com blocos try-catch-finally, await não deve ser utilizada nas seções catch ou finally;
* Métodos Async não podem usar parâmetros "ref" ou "out";
* Construtores de classe não podem ser marcados com async nem podem usar a palavra-chave await;
* Await só pode ser usado em métodos assinalados com async;
* Os assessores Property não podem ser marcados como async nem podem usar await;

A classe FileStream, por exemplo, expõe métodos assíncronos, como WriteAsync e ReadAsync. Eles usam uma implementação que utiliza E/S assíncrona real. Dessa forma, eles não usam um thread enquanto aguardam no disco rígido do seu sistema para ler ou gravar alguns dados.

Quando uma exceção ocorre em um método assíncrono, você normalmente espera uma AggregateException. No entanto, o código gerado ajuda a desembrulhar o AggregateException e lança a primeira de suas exceções internas. Isso torna o código mais intuitivo de usar e mais fácil de depurar.

Outra coisa importante ao trabalhar com código assíncrono é o conceito de um SynchronizationContext, que conecta seu modelo de aplicativo ao modelo de thread. Por exemplo, um aplicativo WPF usa um único thread da interface do usuário e potencialmente vários threads de segundo plano para melhorar a capacidade de resposta e distribuir o trabalho entre várias CPUs. Um aplicativo ASP.NET, no entanto, usa threads do pool de threads que são inicializados com os dados corretos, como usuário e cultura atuais, para atender a solicitações de entrada. O SynchronizationContext abstrai a maneira como esses diferentes aplicativos funcionam e garante que você acabe no thread certo quando precisar atualizar algo na interface do usuário ou processar uma solicitação da Web.

A palavra-chave await garante que o SynchronizationContext atual seja salvo e restaurado quando a tarefa terminar. Ao usar await dentro de um aplicativo WPF, isso significa que, após a conclusão da tarefa, seu programa continua sendo executado no thread da interface do usuário. Em um aplicativo ASP.NET, o código restante é executado em um thread que possui o conjunto de informações culturais, principais e outras informações do cliente.

O exemplo a seguir mostra um aplicativo Windows Forms que baixa um site e coloca o resultado em um Textbox e ConfigureAwait é falso (SynchronizationContext desativado).

HttpClient httpClient = new HttpClient();

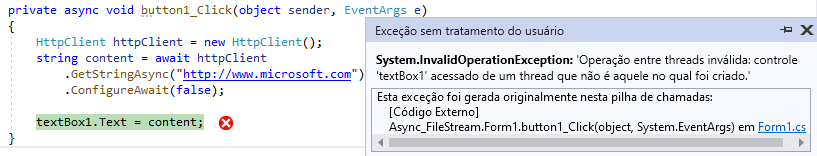
string content = await httpClient

.GetStringAsync("http://www.microsoft.com")

.ConfigureAwait(false);

////Funicona somente com ConfigureAwait(true), SynchronizationContext = true

textBox1.Text = content;



Este exemplo lança uma exceção denominada Cross Threading; a linha textBox1.Text não é executada no thread da interface do usuário devido ao ConfigureAwait(false). Em um ambiente multithread, apenas um thread da interface do usuário pode alterar o valor dos controles da interface do usuário (botão, rótulo, caixa de texto etc.). Se outro thread tentar alterar o valor de um controle de interface do usuário, uma exceção de thread cruzado surgirá porque o Runtime não permitirá que nenhum thread manipule diretamente outros dados de thread.

Para eliminar o erro, neste caso, somente altere para ConfigureAwait(true). Uma outra maneira para alterar os valores do controle da interface do usuário de outros threads é utilizando o método BeginInvoke. Faz isso de uma maneira thread-safe. Requer um delegado, que bloqueia (cria um lock) e informa qual controle da interface do usuário precisa alterar seu valor.

Task task = Task.Run(() =>

{

this.BeginInvoke(new Action(() =>

{

textBox1.Text = content;

}));

});

Desabilitando o SynchronizationContext, seu código tem um desempenho melhor e há casos em que o seu código de continuação possa ser executado em qualquer thread, pois não precisa atualizar a interface do usuário após a conclusão, como ao gravar o conteúdo em um arquivo:

HttpClient httpClient = new HttpClient();

string content = await httpClient

.GetStringAsync("http://www.microsoft.com")

.ConfigureAwait(false);

using (FileStream sourceStream = new FileStream("temp.html",

FileMode.Create, FileAccess.Write, FileShare.None,

4096, useAsync: true))

{

byte[] encodedText = Encoding.Unicode.GetBytes(content);

await sourceStream.WriteAsync(encodedText, 0, encodedText.Length)

.ConfigureAwait(false);

};

A seguir temos um exemplo de código síncrono que acessa um banco de dados e preenche um controle TextBox em um formulário com dados. Esse código é totalmente **síncrono**.

private static void CarregaDados()

{

StringBuilder txtDados = new StringBuilder();

var connectionString = @"Data Source=localhost;Initial Catalog=Cadastro;Integrated Security=True";

string sql = @"select Id,Nome from Alunos";

using (SqlConnection conn = new SqlConnection(connectionString))

{

SqlCommand cmd = new SqlCommand(sql, conn);

conn.Open();

SqlDataReader rdr = cmd.ExecuteReader();

while (rdr.Read())

{

txtDados.Append("\nId: ");

txtDados.Append(rdr.GetValue(0) + "\t\t" + rdr.GetValue(1));

txtDados.Append("\n");

}

Console.WriteLine(txtDados);

}

}

Agora veja como ficou o código do exemplo usando async e await: Este código além de usar async e await também usa os métodos **OpenASync(), ExecuteReaderAsync(), ReadAsync() e GetFieldValueAsync<T>()** da versão 4.5.

private static async void CarregarDadosAssincrono()

{

StringBuilder txtDados = new StringBuilder();

var connectionString = @"Data Source=localhost;Initial Catalog=Cadastro;Integrated Security=True";

string sql = @"select Id,Nome from Alunos";

using (SqlConnection conn = new SqlConnection(connectionString))

{

SqlCommand cmd = new SqlCommand(sql, conn);

await conn.OpenAsync();

using (SqlDataReader rdr = await cmd.ExecuteReaderAsync())

{

while (await rdr.ReadAsync())

{

txtDados.Append("\nId: ");

txtDados.Append(await rdr.GetFieldValueAsync<int>(0) + "\t\t" + await rdr.GetFieldValueAsync<string>(1));

txtDados.Append("\n");

}

Console.WriteLine(txtDados);

}

}

}

É importante decidir antes de criar um **SqlDataReader** se você precisa usar o modo de acesso não-sequencial ou sequencial. Na maioria dos casos, utilizar o modo de acesso padrão (não-sequencial) é a melhor opção, pois permite um modelo de programação mais fácil (você pode acessar qualquer coluna em qualquer ordem), e você obterá um melhor desempenho usando **ReadAsync**.

No entanto, desde que o modo de acesso não sequencial tem de armazenar os dados para toda a linha, ela pode causar problemas se você estiver lendo uma grande coluna do servidor (como varbinary (MAX), varchar (max), nvarchar (MAX) ou XML). Neste caso, usando o modo de acesso seqüencial lhe permitirá transmitir estas grandes colunas ao invés de ter que colocar a coluna inteira na memória.

É também uma boa ideia chamar ReadAsync no modo não-seqüencial isso vai ler todos os dados da coluna, o que pode abranger vários pacotes, permitindo o acesso mais rápido aos valores da coluna. A seguir uma breve descrição dos métodos usados:

|  |  |
| --- | --- |
| **Métodos** | **Descrição** |
| OpenASync() | Esta é a versão assíncrona do método Open. Os provedores devem ser sobrescritos com uma implementação adequada. A implementação padrão invoca a chamada Open síncrona e retorna uma tarefa concluída. Exceções geradas por Open serão comunicadas através da propriedade Exception Task retornada. Não invoque outros métodos e propriedades do objeto DbConnection até que a tarefa retornada esteja completa. |
| ExecuteReaderAsync() | Inicia a execução assíncrona da instrução Transact-SQL ou procedimento armazenado que é descrito por este SqlCommand. |
| ReadAsync() | Uma versão assíncrona de método Read, que avança o leitor para o próximo registro em um conjunto de resultados. Este método chama ReadAsync com CancellationToken.None. |
| GetFieldValueAsync<T>() | Obtém o valor especifica pela coluna do tipo de forma assíncrono. |

**PLINQ - Parallel Language Integrated Query**

PLINQ é a versão paralela do LINQ. Você pode usá-lo em objetos para potencialmente transformar uma consulta seqüencial em paralela sobre todos os tipos de dados. Isso significa que as consultas podem ser executadas em vários threads, particionando a fonte de dados em threads. Cada threadé executado em threads de trabalho separados em paralelo em vários processadores. Os métodos de extensão para usar o PLINQ são definidos na classe System.Linq.ParallelEnumerable. Versões paralelas de operadores LINQ, como Where, Select, SelectMany, GroupBy, Join, OrderBy, Skip e Take, podem ser usadas. Os seguintes métodos comuns para ajudar no paralelismo:

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| AsParallel () | Divide a fonte de dados em threads em vários threads |
| AsSequential () | Especifique que a consulta deve ser executada sequencialmente |
| AsOrdered () | Especifique que a consulta deve preservar a ordem dos dados |
| AsUnordered() | a consulta não deve preservar a ordem dos dados |
| ForAll () | Processa o resultado em paralelo |

O tempo de execução determina se faz sentido transformar sua consulta em paralela. Ao fazer isso, ele gera objetos Task e começa a executá-los. Se você deseja forçar o PLINQ a uma consulta paralela, pode usar o método WithExecutionMode e especificar que ele sempre deve executar a consulta em paralelo.

Você também pode limitar a quantidade de paralelismo usada com o método WithDegreeOfParallelism. Você passa esse método para um número inteiro que representa o número de processadores que deseja usar. Normalmente, o PLINQ usa todos os processadores (até 64), mas você pode limitá-lo com este método, se desejar.

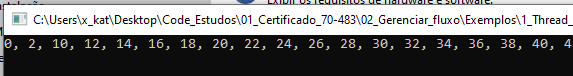
Uma coisa a ter em mente é que o processamento paralelo não garante nenhuma ordem específica. Os resultados variam dependendo da quantidade de CPUs disponíveis. Ao usar o PLINQ, você pode usar o operador ForAll para iterar sobre uma coleção, quando a iteração também pode ser feita de maneira paralela. Ao contrário do foreach, o ForAll não precisa de todos os resultados antes de iniciar a execução. Neste exemplo, o ForAll remove, no entanto, qualquer ordem de classificação especificada. O código a seguir mostra como isso funciona.

var numbers = Enumerable.Range(0, 50);

var parallelResult = numbers.AsParallel().WithDegreeOfParallelism(10)

.Where(i => i % 2 == 0);

parallelResult.ForAll(e => Console.Write(e + ", "));



Se você deseja garantir que os resultados sejam ordenados, você pode adicionar o operador AsOrdered. Sua consulta ainda é processada em paralelo, mas os resultados são armazenados em buffer e classificados. Se você possui uma consulta complexa que pode se beneficiar do processamento paralelo, mas também possui algumas partes que devem ser feitas sequencialmente, é possível usar o AsSequential para impedir que a consulta seja processada em paralelo. O exemplo abaixo mostra como você pode usar o operador AsSequential para garantir que o método Take retorne ordenada.

var parallelOrdered = numbers.AsParallel().AsOrdered()

.Where(i => i % 2 == 0).AsSequential();

foreach (int i in parallelResult.Take(5))

Console.WriteLine(i);

Obviamente, pode acontecer que algumas das operações em sua consulta paralela gerem uma exceção. O .NET Framework lida com isso agregando todas as exceções em um AggregateException. Esta exceção expõe uma lista de todas as exceções que ocorreram durante a execução paralela. O exemplo abaixo mostra como você pode lidar com isso.

public static bool IsEven(int i)

{

if (i % 10 == 0) throw new ArgumentException("i");

return i % 2 == 0;

}

var numbers = Enumerable.Range(0, 20);

try

{

var parallelResult = numbers.AsParallel()

.Where(i => IsEven(i));

parallelResult.ForAll(e => Console.WriteLine(e));

}

catch (AggregateException e)

{

Console.WriteLine("There where {0} exceptions", e.InnerExceptions.Count);

}

Como você pode ver, duas exceções foram lançadas durante o processamento dos dados. Você pode inspecionar essas exceções percorrendo a propriedade InnerExceptions.

**CONCURRENT COLLECTIONS**

Em C#, temos as coleções definidas em System.Collections e System.Collections.Generic. As coleções genéricas são de tipo seguro, o que significa que, em tempo de compilação, podemos fazer uma coleção de qualquer tipo. Mas essas coleções não são seguras para threads. Antes do .NET 4, se você precisava de uma coleção segura para threads, precisava implementá-la por conta própria. Eles se tornam vulneráveis ​​quando vários threads podem manipular os mesmos dados ao mesmo tempo. Ao trabalhar em um ambiente multithread, você precisa garantir que não está manipulando dados compartilhados ao mesmo tempo sem sincronizar o acesso. Em um ambiente multithread, vários threads podem acessar os mesmos dados ao mesmo tempo para ler/adicionar/editar. Esses dados não são seguros para threads e se tornam vulneráveis ​​a vários threads.

O .NET Framework oferece algumas classes de coleção criadas especificamente para uso em ambientes simultâneos, que é o que você tem quando usa multithreading. Essas coleções são seguras para threads, o que significa que elas usam internamente a sincronização para garantir que possam ser acessadas por vários threads ao mesmo tempo. Coleções genéricas também podem se tornar seguras para threads se forem usadas em uma instrução de bloqueio adequada, mas bloquear a coleção inteira para adicionar/remover um item pode ser um grande problema de desempenho.

As Concurrent Collections definidas no namespace System.Collections.Concurrent, são as seguintes:

* **ConcurrentDictionary<TKey, T>**: dicionário seguro de thread em pares de valores-chave
* **BlockingCollection<T>:** fornece um padrão de consumidor de produtor clássico. Implementa a interface IProducerConsumerCollection<T>, fornecendo recursos de bloqueio e delimitação.
* **ConcurrentBag<T>**: implementação segura de thread de uma coleção não ordenada, possui o método Add() para adicionar um item e o método TryTake() para remover e retornar o item
* **ConcurrentQueue<T>**: estrutura de dados FIFO segura para thread, possui o método Enque() para enfileirar um item e o método TryDeque() para remover e retornar o primeiro item.
* **ConcurrentStack<T>**: estrutura de dados LIFO segura para threads, possui o método Push() para enviar um item e o método TryPop() para remover e retornar o último item.

Da lista acima, o ConcurrentDictionary pode ser usado como uma coleção de uso geral, enquanto outros são usados principalmente em cenários produtor-consumidor (ou seja, threads dedicados para adicionar e excluir).

**ConcurrentDictionary<TKey, T>**

Um ConcurrentDictionary armazena pares de chave e valor de maneira segura para threads. Você pode usar métodos para adicionar e remover itens e atualizar itens no local, se existirem. Ao trabalhar com um ConcurrentDictionary, você tem métodos que podem adicionar, obter e atualizar itens atomicamente. Uma operação atômica significa que será iniciada e finalizada como uma única etapa sem que outros threads interfiram. A tabela abaixo mostra os métodos que você pode usar em um ConcurrentDictionary.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| TryAdd | Se outro thread tentar adicionar um novo valor de chave que já foi adicionado por outro thread, ele ignora a iteração e move o controle para a próxima iteração. |
| TryUpdate | Verifica se o valor atual é igual ao valor existente antes de atualizá-lo. |
| AddOrUpdate | garante que um item seja adicionado se não estiver lá e atualizado para um novo valor, se estiver. |
| GetOrAdd | obtém o valor atual de um item, se estiver disponível; caso contrário, ele adiciona o novo valor usando um método de fábrica. |

var dict = new ConcurrentDictionary<string, int>();

if (dict.TryAdd("k1", 42)) Console.WriteLine("Added");

if (dict.TryUpdate("k1", 21, 42)) Console.WriteLine("42 updated to 21");

dict["k1"] = 42; // Overwrite unconditionally

int r1 = dict.AddOrUpdate("k1", 3, (s, i) => i \* 2);

int r2 = dict.GetOrAdd("k2", 3);

Por exemplo, ao usar uma coleção genérica em vários threads, Tsk1 e Tsk2 tentaram manipular a chave do dicionário; portanto, ocorre um erro.

Dictionary<int, int> dic = new Dictionary<int, int>();

Task tsk1 = Task.Run(() =>

{

for (int i = 0; i < 100; i++)

dic.Add(i, i + 1);

});

Task tsk2 = Task.Run(() =>

{

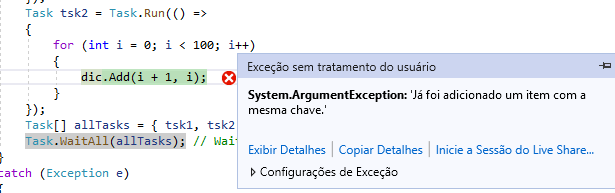
for (int i = 0; i < 100; i++)

dic.Add(i + 1, i);

});

Task[] allTasks = { tsk1, tsk2 };

Task.WaitAll(allTasks); // Wait for all tasks



Já com ConcurrentDictionary <K, V> que é uma coleção segura para threads; no snippet de código abaixo, ele impede que vários threads funcionem no mesmo valor de chave. Dessa forma, nenhum conflito ocorre e, portanto, o programa é executado com êxito.

ConcurrentDictionary<int, int> dic = new ConcurrentDictionary<int, int>();

Task tsk1 = Task.Run(() =>

{

for (int i = 0; i < 100; i++)

dic.TryAdd(i, i + 1);

});

Task tsk2 = Task.Run(() =>

{

for (int i = 0; i < 100; i++)

dic.TryAdd(i + 1, i);

});

Task[] allTasks = { tsk1, tsk2 };

Task.WaitAll(allTasks); // Wait for all tasks

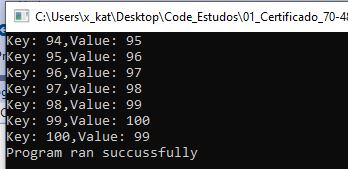
foreach (var item in dic)

{

Console.WriteLine($"Key: {item.Key},Value: {item.Value}");

}

Console.WriteLine("Program ran succussfully");



**BlockingCollection <T>**

Esta coleção é segura para threads para adicionar e remover dados. A remoção de um item da coleção pode ser bloqueada até que os dados estejam disponíveis. A adição de dados é rápida, mas você pode definir um limite máximo. Se esse limite for atingido, a adição de um item bloqueará o thread de chamada até que haja espaço. BlockingCollection é, na realidade, um invólucro em torno de outros tipos de coleção. Se você não fornecer instruções específicas, ele usará o ConcurrentQueue por padrão.

Uma coleção regular explode ao ser usada em um cenário multithread porque um item pode ser removido por um thread enquanto o outro thread está tentando lê-lo. O código abaixo mostra um exemplo de uso de um BlockingCollection. Uma tarefa ouve novos itens sendo adicionados à coleção. Ele bloqueia se não houver itens disponíveis. A outra tarefa adiciona itens à coleção.

BlockingCollection<string> col = new BlockingCollection<string>();

Task read = Task.Run(() =>

{

while (true)

{

//Se comentar essa linha imprimira infinitamente o "oi"

Console.WriteLine(col.Take());

}

});

Task write = Task.Run(() =>

{

while (true)

{

string s = Console.ReadLine();

if (string.IsNullOrWhiteSpace(s)) break;

col.Add(s);

}

});

write.Wait();

O programa termina quando o usuário não insere dados. Até isso, todas as strings inseridas são adicionadas pela Task write e removidas pela Task read. Você pode usar o método CompleteAdding para sinalizar para o BlockingCollection que nenhum outro item será adicionado. Se outros threads aguardarem novos itens, eles não serão mais bloqueados. A seguir outro exemplo onde limita-se o tamanho máximo para 10 itens e utiliza o CompleteAdding para finalizaro processo:

Console.WriteLine("BlockingCollection Com Limite");

// A blocking collection that can hold no more than 10 items at a time.

var numberCollection = new BlockingCollection<string>(10);

Task read\_lim = Task.Run(() =>

{

while (!numberCollection.IsCompleted)

{

Console.WriteLine("Read " + numberCollection.Take());

}

Console.WriteLine("\r\nNo more items to take.");

});

// A simple blocking producer with no cancellation.

Task write\_lim = Task.Run(() =>

{

foreach (int i in Enumerable.Range(1, 10))

{

Console.WriteLine("adding " + i);

numberCollection.Add(i.ToString());

}

numberCollection.CompleteAdding();

});

write\_lim.Wait();

read\_lim.Wait();

Console.WriteLine("FIM BlockingCollection Com Limite");

Você pode até remover as instruções while (true) do Task read, usando o método GetConsumingEnumerable, você obtém um IEnumerable que bloqueia até encontrar um novo item. Dessa forma, você pode usar um foreach com seu BlockingCollection para enumerá-lo e não precisa usar mais o método Take().

BlockingCollection<string> col = new BlockingCollection<string>();

Task read = Task.Run(() =>

{

foreach (string v in col.GetConsumingEnumerable())

Console.WriteLine(v);

});

**ConcurrentBag<T>:**

Um ConcurrentBag é apenas um saco de itens. Permite duplicatas e não possui uma ordem específica. Os métodos importantes são Add, TryTake e TryPeek.

ConcurrentBag<int> bag = new ConcurrentBag<int>();

bag.Add(42);

bag.Add(21);

int result;

if (bag.TryTake(out result))

Console.WriteLine("TryTake: " + result);

if (bag.TryPeek(out result))

Console.WriteLine("There is a next item: {0}", result);

Um aspecto a ter em mente é que o método TryPeek não é muito útil em um ambiente multithread. Pode ser que outro thread remova o item antes que você possa acessá-lo. ConcurrentBag também implementa IEnumerable <T>, para que você possa iterar sobre ele. Essa operação é tornada segura para threads, fazendo um instantâneo da coleção quando você começa a iterá-la, para que os itens adicionados à coleção depois que você começou a iterá-la não fiquem visíveis. O exemplo abaixo mostra isso na prática.

ConcurrentBag<int> bag2 = new ConcurrentBag<int>();

Task.Run(() =>

{

bag2.Add(42);

Thread.Sleep(1000); // Se comentar aqui, aparece 0 21 e 42

bag2.Add(21);

});

Task.Run(() =>

{

foreach (int i in bag2)

Console.WriteLine(i);

}).Wait();

Esse código exibe apenas 42 porque o outro valor é adicionado após o início da iteração sobre a bolsa.

**ConcurrentQueue<T>**

Uma fila é uma coleção FIFO (primeiro a entrar, primeiro a sair). ConcurrentQueue oferece os métodos Enqueue e TryDequeue para adicionar e remover itens da coleção. Ele também possui um método TryPeek e implementa IEnumerable criando um instantâneo dos dados. A código abaixo mostra como usar um ConcurrentQueue.

ConcurrentQueue<int> queue = new ConcurrentQueue<int>();

queue.Enqueue(42);

int result;

if (queue.TryDequeue(out result))

Console.WriteLine("Dequeued: {0}", result);

O exemplo a seguir cria uma fila contendo todos os números inteiros entre um e um milhão. Duas tarefas paralelas são iniciadas, cada uma executando um while somando os valores até a fila se esgotar; nesse momento. Cada iteração do loop remove da fila um valor e o adiciona a um total. No entanto, ao final, uma exceção é lançada.

IEnumerable<int> numbers = Enumerable.Range(1, 1000000);

Queue<int> \_queued = new Queue<int>(numbers);

long \_total = 0;

Task task1 = Task.Run(() =>

{

while (true)

{

Interlocked.Add(ref \_total, \_queued.Dequeue());

}

});

Task task2 = Task.Run(() =>

{

while (true)

{

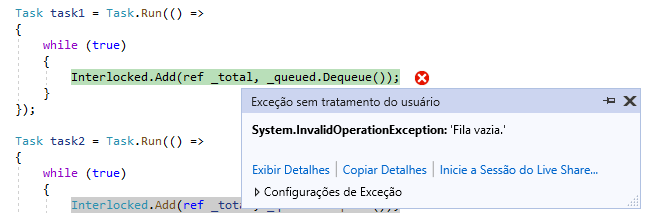
Interlocked.Add(ref \_total, \_queued.Dequeue());

}

});

Task.WaitAll(task1, task2);

Console.WriteLine("Total: {0}", \_total);



Vamos reescrever o exemplo anterior para usar uma fila simultânea. Isso requer apenas algumas modificações. Primeiro, precisamos alterar o tipo do objeto da fila. Em seguida, precisamos usar o TryDequeue em vez do Dequeue e adicionar o valor do parâmetro de saída ao total. Não precisamos mais esperar que uma exceção seja lançada para saber que esgotamos a fila. Em vez disso, podemos usar o valor de retorno do TryDequeue, interrompendo o loop quando o método retornar false.

IEnumerable<int> numbers = Enumerable.Range(1, 1000000);

ConcurrentQueue<int> \_queued = new ConcurrentQueue<int>(numbers);

long \_total = 0;

int value;

Task task1 = Task.Run(() =>

{

while (\_queued.TryDequeue(out value))

{

Interlocked.Add(ref \_total, value);

}

});

Task task2 = Task.Run(() =>

{

while (\_queued.TryDequeue(out value))

{

Interlocked.Add(ref \_total, value);

}

});

Task.WaitAll(task1, task2);

Console.WriteLine("Total: {0}", \_total); //Total: 500000288629

O código atualizado é mostrado abaixo. No final do programa, o total é emitido para o console. Você deve ver um total de 500.000.500.000. No entanto, se você estiver usando um computador com mais de um núcleo de processador, provavelmente verá um resultado diferente.

A seguir iremos avaliar prós e contras de diferentes abordagens, começaremos com a Queue básica e avançaremos para uma solução final com ConcurrentQueue.

**Fila regular com thread único**

No exemplo abaixo, a fila genérica é usada para armazenar informações do pedido. Além disso, o método GetOrders é chamado na maneira de sincronização regular.

private static void GetOrders(string custName, Queue<string> phoneOrders)

{

for (int i = 0; i < 3; i++)

{

Thread.Sleep(100);

string order = string.Format("{0} needs {1} phones", custName, i + 5);

phoneOrders.Enqueue(order);

}

}

var phoneOrders = new Queue<string>();

GetOrders("Prakash", phoneOrders);

GetOrders("Aradhana", phoneOrders);

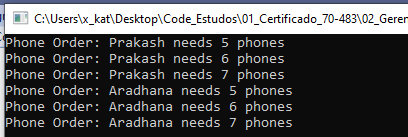
foreach (var order in phoneOrders)

{

Console.WriteLine("Phone Order: {0}", order);

}

Como o método GetOrders é chamado em sincronia ou um após o outro, a saída também é impressa de maneira semelhante (ou seja, primeiro Prakash e depois Aradhana).



**Fila regular com mais de um thread**

Agora, vamos fazer a pequena alteração no código anterior, tornando-o assíncrono. Para isso, usamos uma tarefa que chamará GetOrders por dois threads diferentes.

var phoneOrders = new Queue<string>();

Task t1 = Task.Run(() => GetOrders("Prakash", phoneOrders));

Task t2 = Task.Run(() => GetOrders("Aradhana", phoneOrders));

Task.WaitAll(t1, t2);

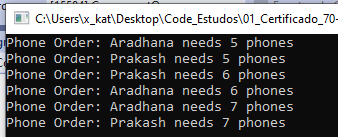
foreach (var order in phoneOrders)

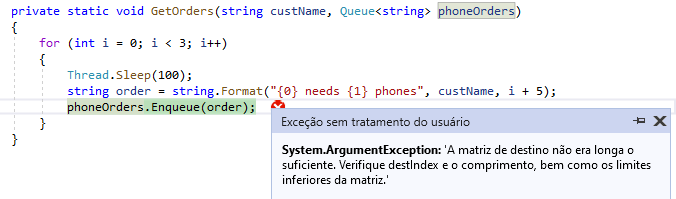
{

Console.WriteLine("Phone Order: {0}", order);

}

Ocorre uma exceção porque o método Enqueue da Fila não foi projetado para funcionar com mais de um thread paralelamente.O Multi-threading com a fila regular é imprevisível. Pode funcionar em alguns casos, mas se você tentar várias vezes, provavelmente receberá uma exceção, como acima.





**Fila regular com bloqueio manual e mais de um thread**

A solução é ter algum tipo de sincronização de threads, manualmente ou fora da caixa. A seguir, vamos ver a maneira manual com uso da palavra-chave de **lock**.

static object lockObj = new object();

private static void GetOrdersWithLock(string custName, Queue<string> phoneOrders)

{

for (int i = 0; i < 3; i++)

{

Thread.Sleep(100);

string order = string.Format("{0} needs {1} phones", custName, i + 5);

lock (lockObj)

{

phoneOrders.Enqueue(order);

}

}

}

var phoneOrders = new Queue<string>();

Task t1 = Task.Run(() => GetOrdersWithLock("Prakash", phoneOrders));

Task t2 = Task.Run(() => GetOrdersWithLock("Aradhana", phoneOrders));

Task.WaitAll(t1, t2);

foreach (var order in phoneOrders)

{

Console.WriteLine("Phone Order: {0}", order);

}

Portanto, não há exceção neste momento, depois de colocar o bloqueio no método Enqueue Mas e se o Enqueue for chamado várias vezes, você teria que usar a instrução lock em várias partes.

**ConcurrentQueue com mais de um thread**

Como você pode ver, para gerenciar a simultaneidade, não precisamos mais de bloqueio manual. É mais útil em situações em que, em um ambiente multithread, estamos lidando com métodos de fila em vários locais; e, colocar o bloqueio manual em todo lugar pode tornando um código impossível de manter.

private static void GetOrders(string custName, object phoneOrders)

{

for (int i = 0; i < 3; i++)

{

Thread.Sleep(100);

string order = string.Format("{0} needs {1} phones", custName, i + 5);

if (phoneOrders is ConcurrentQueue<string>)

(phoneOrders as ConcurrentQueue<string>).Enqueue(order);

else if (phoneOrders is Queue<string>)

(phoneOrders as Queue<string>).Enqueue(order);

}

}

var phoneOrders = new ConcurrentQueue<string>();

Task t1 = Task.Run(() => GetOrders("Prakash", phoneOrders));

Task t2 = Task.Run(() => GetOrders("Aradhana", phoneOrders));

Task.WaitAll(t1, t2);

foreach (var order in phoneOrders)

{

Console.WriteLine("Phone Order: {0}", order);

}

Os métodos mais usados do ConcurrentQueue são:

* TryPeek: que buscará o elemento desde o início da fila sem removê-lo
* TryDequeue: que remove o elemento do início da fila,

Eles definem o elemento excluído na variável out e retorna true, caso contrário, retorna false. Vamos dar uma olhada no código e como usá-lo.

Console.WriteLine("Total orders before Dequeue/TryPeek are: {0}", phoneOrders.Count);

string myOrder;

if (phoneOrders.TryPeek(out myOrder)) //TryPeek

Console.WriteLine("Order \"{0}\" has been retrieved", myOrder);

//TryDequeue, Deletes the item from beginning of queue.

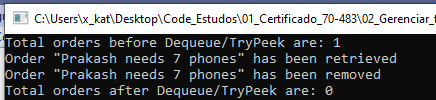
if (phoneOrders.TryDequeue(out myOrder))

Console.WriteLine("Order \"{0}\" has been removed", myOrder);

else

Console.WriteLine("Order queue is empty", myOrder);

Console.WriteLine("Total orders after Dequeue/TryPeek are: {0}", phoneOrders.Count);



**ConcurrentStack<T>**

As implementações de ConcurrentStack são similares ao do ConcurrentQueue, com a diferença que a coleção ConcurrentStack é uma pilha LIFO (último a entrar, primeiro a sair). O ConcurrentStack possui dois métodos importantes: Push e TryPop. Push é usado para adicionar um item à pilha; O TryPop tenta obter um item da pilha. Você nunca pode ter certeza se há itens na pilha porque vários threads podem estar acessando sua coleção ao mesmo tempo. Você também pode adicionar e remover vários itens de uma vez usando PushRange e TryPopRange. Quando você enumera a coleção, é tirado um instantâneo. O exemplo a seguir mostra como esses métodos funcionam.

ConcurrentStack<int> stack = new ConcurrentStack<int>();

stack.Push(42);

int result;

if (stack.TryPop(out result))

Console.WriteLine("Popped: {0}", result);

stack.PushRange(new int[] { 1, 2, 3 });

int[] values = new int[2];

stack.TryPopRange(values);

foreach (int i in values)

Console.WriteLine(i);

Uma impementação semelhante da ConcurrentQueue<T> que vimos em exemplo anterior. Primeiro, uma pilha é inicializada e preenchida com todos os valores inteiros entre um e um milhão. Todo valor é então retirado da pilha e somado. O processo continua até que uma exceção seja lançada quando a pilha estiver esgotada. Devemos esperar que o total final seja de 500.000.500.000. No entanto, estamos usando duas tarefas paralelas para processar a pilha e a pilha básica <T> não é segura para threads. Isso leva a condições de corrida e a um resultado incorreto ao usar um computador com mais de um núcleo de processador.

IEnumerable<int> numbers = Enumerable.Range(1, 1000000);

Stack<int> \_stack = new Stack<int>(numbers);

long \_total = 0;

int value;

Task task1 = Task.Run(() =>

{

while (true)

{

Interlocked.Add(ref \_total, \_stack.Pop());

}

});

Task task2 = Task.Run(() =>

{

while (true)

{

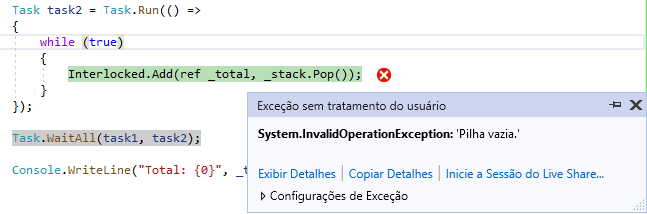
Interlocked.Add(ref \_total, \_stack.Pop());

}

});

Task.WaitAll(task1, task2);

Console.WriteLine("Total: {0}", \_total);



Como no exemplo do artigo ConcurrentQueue <T>, você pode resolver esse problema com a instrução lock. No entanto, a classe ConcurrentStack <T> torna muito mais simples.

IEnumerable<int> numbers = Enumerable.Range(1, 1000000);

ConcurrentQueue<int> \_queued = new ConcurrentQueue<int>(numbers);

long \_total = 0;

int value;

Task task1 = Task.Run(() =>

{

while (\_queued.TryDequeue(out value))

{

Interlocked.Add(ref \_total, value);

}

});

Task task2 = Task.Run(() =>

{

while (\_queued.TryDequeue(out value))

{

Interlocked.Add(ref \_total, value);

}

});

Task.WaitAll(task1, task2);

Console.WriteLine("Total: {0}", \_total); // Total: 499933914565

**Interface IProducerConsumerCollection<T>**

Define métodos para manipular coleções thread-safe destinadas ao uso de produtor/consumidor. Essa interface fornece uma representação unificada de coleções de produtor/consumidor para que abstrações de níveis mais altos, como o BlockingCollection<T>, possam usar a coleção como o mecanismo de armazenamento subjacente.

Com exceção da classe ConcurrentDictionary, todas as classes de coleção simultânea implantadas pela Microsoft implementam a interface IProducerConsumerCollection. Essa interface requer uma classe que a implemente para fornecer os métodos abaixo:

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| CopyTo | Copia os elementos do objeto IProducerConsumerCollection em uma matriz, iniciando no local especificado. |
| ToArray | Retorna uma nova matriz que contém todos os elementos no IProducerConsumerCollection. |
| TryAdd | Tenta adicionar um objeto ao IProducerConsumerCollection. |
| TryTake | Tenta remover e retornar um objeto do IProducerConsumerCollection. |

O exemplo a seguir mostra uma estrutura de dados de pilha que implementa a interface:

public class SafeStack<T> : IProducerConsumerCollection<T>

{

// Used for enforcing thread-safety

private object m\_lockObject = new object();

// We'll use a regular old Stack for our core operations

private Stack<T> m\_sequentialStack = null;

#region Support for ICollection

public int Count { get { return m\_sequentialStack.Count; } }

public object SyncRoot { get { return m\_lockObject; } }

public bool IsSynchronized { get { return true; } }

#endregion

#region Constructors

public SafeStack()

{

m\_sequentialStack = new Stack<T>();

}

public SafeStack(IEnumerable<T> collection)

{

m\_sequentialStack = new Stack<T>(collection);

}

#endregion

#region IProducerConsumerCollection(T)

public void CopyTo(T[] array, int index)

{

lock (m\_lockObject) m\_sequentialStack.CopyTo(array, index);

}

public void CopyTo(Array array, int index)

{

lock (m\_lockObject) ((ICollection)m\_sequentialStack).CopyTo(array, index);

}

public T[] ToArray()

{

T[] rval = null;

lock (m\_lockObject) rval = m\_sequentialStack.ToArray();

return rval;

}

public bool TryAdd(T item)

{

Push(item);

return true; // Push doesn't fail

}

public bool TryTake(out T item)

{

return TryPop(out item);

}

// Safe Push/Pop support

public void Push(T item)

{

lock (m\_lockObject) m\_sequentialStack.Push(item);

}

public bool TryPop(out T item)

{

bool rval = true;

lock (m\_lockObject)

{

if (m\_sequentialStack.Count == 0) { item = default(T); rval = false; }

else

{

item = m\_sequentialStack.Pop();

}

}

return rval;

}

#endregion

#region IEnumerable

// Support for IEnumerable

IEnumerator IEnumerable.GetEnumerator()

{

return ((IEnumerable<T>)this).GetEnumerator();

}

// Support for IEnumerable(T)

public IEnumerator<T> GetEnumerator()

{

// The performance here will be unfortunate for large stacks,

// but thread-safety is effectively implemented.

Stack<T> stackCopy = null;

lock (m\_lockObject) stackCopy = new Stack<T>(m\_sequentialStack);

return stackCopy.GetEnumerator();

}

#endregion

}

SafeStack<int> stack = new SafeStack<int>();

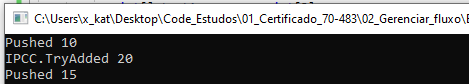
IProducerConsumerCollection<int> ipcc = (IProducerConsumerCollection<int>)stack;

// Test Push()/TryAdd()

stack.Push(10); Console.WriteLine("Pushed 10");

ipcc.TryAdd(20); Console.WriteLine("IPCC.TryAdded 20");

stack.Push(15); Console.WriteLine("Pushed 15");



Gerenciar multithreading

* Sincronizar recursos; implementar bloqueio; cancelar uma tarefa de execução longa; implementar métodos thread-safe para manipular condições de corrida

Embora o multithreading possa oferecer muitas vantagens, não é fácil escrever um aplicativo multithread. Podem ocorrer problemas quando threads diferentes acessam alguns dados compartilhados, tais como:

* Não é tão fácil de seguir e entender.
* Não é tão previsível; embora, você precise garantir o mesmo resultado.
* Não é tão fácil depurar, tornando mais difícil encontrar bugs.
* É mais difícil de testar.

Como afirmado anteriormente, um dos problemas mais comuns é chamado de condição de corrida. Isso acontece quando dois threads tentam atualizar os mesmos dados. O que deve acontecer quando ambos tentam mudar algo ao mesmo tempo? Para fazer isso funcionar com sucesso, é importante sincronizar recursos.

**SINCRONIZAÇÃO DE VARIÁVEIS NO MULTITHREADING**

Em um ambiente multithreading, a mesma variável pode ser acessada por dois ou mais threads. Se a operação executada em uma variável compartilhada for atômica ou segura para threads, ela produzirá um resultado preciso. Se a operação não for atômica ou não for segura para threads, ela produzirá resultados imprecisos.

Na operação atômica, apenas um único thread de cada vez pode executar uma única instrução e produzir resultados precisos; enquanto, em uma operação não atômica, mais de um thread está acessando e manipulando o valor de uma variável compartilhada, o que produz um resultado impreciso (por exemplo, se um thread está lendo um valor e o outro thread ao mesmo tempo está editando isto).

int num = 0;

int length = 500000;

//Run on separate thread of threadpool

Task tsk = Task.Run(() =>

{

for (int i = 0; i < length; i++)

{

num = num + 1;

}

});

//Run on Main Thread

for (int i = 0; i < length; i++)

{

num = num - 1;

}

tsk.Wait();

Console.WriteLine(num); //-6674 OU o OU 655

O trecho de código acima fornece resultados imprecisos porque dois threads estão acessando e manipulando o valor de "num" ao mesmo tempo. A declaração "num = num + 1;" é realmente uma combinação de mais de uma declaração; primeiro ele lerá o valor atual de "num", depois adiciona ou subtrai 1 do seu valor atual e o atribuirá a "num".

Imagine se a Main thread leu o valor de num = 6, mas o outro thread leu o valor de num = 3. Quando a Main thread diminui o valor de "num", ela se torna 5. Mas o outro thread já leu o valor de num = 3; quando incrementado, o valor de num torna-se "4", o que é totalmente errado. Isso ocorre porque a operação não é atômica. Consiste em uma leitura e uma gravação que acontecem em momentos diferentes. É por isso que o acesso aos dados com os quais você trabalha precisa ser sincronizado, para que você possa prever com segurança como seus dados são afetados.

As construções de sincronização podem ser divididas em quatro categorias:

1. **Métodos de bloqueio simples**

Eles aguardam a conclusão de outro thread ou o período de tempo decorrido. Os métodos são:

* **Thread.Sleep**
* **Thread.Join**
* **Task.Wait/WaitAll**

1. **Construções de bloqueio (locks)**

Isso limita o número de threads que podem executar alguma atividade ou executar uma seção de código por vez. Construções de bloqueio exclusivas são as mais comuns - elas permitem apenas um thread de cada vez e permitem que os threads concorrentes acessem dados comuns sem interferir entre si. As construções de bloqueio exclusivas padrão são:

* **Palavra-chave lock**
* **Monitor (Enter/Exit)**
* **Mutex**
* **SpinLock**

As construções de bloqueio não-exclusivas são:

* **Semaphore/SemaphoreSlim** **(introduced in Framework 4.0)**
* **ReaderWriterLockSlim**

1. **Construções de sinalização**

Isso permite que um thread pause até receber uma notificação de outro thread, evitando a necessidade de pesquisas ineficientes. Existem dois dispositivos de sinalização comumente usados:

* **EventWaitHandle (AutoResetEvent, ManualResetEvent)**
* **Monitor (métodos Wait/Pulse/PulseAll)**

No Framework 4.0 apresenta as classes:

* **CountdownEvent**
* **Barrier**

1. **Construções de sincronização sem bloqueio**

Eles protegem o acesso a um campo comum, chamando as primitivas do processador. O CLR e o C# fornecem as seguintes construções sem bloqueio:

* **Thread.MemoryBarrier,**
* **Thread.VolatileRead,**
* **Thread.VolatileWrite**
* **Palavra-chave Volatile**
* **Interlock**

**Dead Lock**

Em um ambiente multithread, quando ocorre um deadlock pode ocorrer; ele congela o aplicativo porque duas ou mais atividades aguardam a conclusão uma da outra, fazendo com que nenhum deles seja concluído.

//used as lock objects

object thislockA = new object();

object thislockB = new object();

Task tsk1 = Task.Run(() =>

{

lock (thislockA)

{

Console.WriteLine("thislockA of tsk1");

lock (thislockB)

{

Console.WriteLine("thislockB of tsk2");

Thread.Sleep(100);

}

}

});

Task tsk2 = Task.Run(() =>

{

lock (thislockB)

{

Console.WriteLine("thislockB of tsk2");

lock (thislockA)

{

Console.WriteLine("thislockA of tsk2");

Thread.Sleep(100);

}

}

});

Task[] allTasks = { tsk1, tsk2 };

Task.WaitAll(allTasks); // Wait for all tasks

Console.WriteLine("Program executed succussfully");

Aqui está como o aplicativo ficou congelado.

1. Tsk1 adquire o bloqueio "thislockA".
2. Tsk2 adquire o bloqueio "thislockB".
3. O Tsk1 tenta adquirir o bloqueio "thislockB", mas ele já está retido pelo Tsk2 e, portanto, o Tsk1 bloqueia até que "thislockB" seja liberado.
4. Tsk2 tenta adquirir o bloqueio "thislockA", mas é mantido pelo Tsk1 e, portanto, o Tsk2 bloqueia até que "thislockA" seja liberado. Nesse ponto, os dois threads estão bloqueados e nunca serão ativados. Portanto, o aplicativo congelou.

Para impedir que um aplicativo congele, é importante usar uma declaração de lock com cuidado; caso contrário, você atirará no seu próprio pé.

Abaixo iremos examinar brevemente as maneiras comuns de lidar com variáveis ​​de sincronização em um ambiente multithread.

**MÉTODOS DE BLOQUEIO SIMPLES**

**Join and Sleep**

Você pode esperar que outro thread termine chamando seu método Join. Por exemplo:

static void Main(string[] args)

{

Thread t = new Thread(Go);

t.Start();

t.Join();

Console.WriteLine("Thread t has ended!");

}

static void Go()

{

for (int i = 0; i < 1000; i++) Console.Write("y");

}

Isso imprime "y" 1.000 vezes, seguido de "O tópico t terminou!" imediatamente depois. Você pode incluir um tempo limite ao chamar Join, em milissegundos ou como TimeSpan. Em seguida, retorna true se o thread terminou ou false se o tempo limite expirou.

Thread.Sleep pausa o thread atual por um período especificado:

Thread.Sleep(TimeSpan.FromHours(1)); // sleep for 1 hour

Thread.Sleep(500); // sleep for 500 milliseconds

Enquanto aguarda um Sleep ou Join, um thread é bloqueado e, portanto, não consome recursos da CPU.

Thread.Sleep (0) renuncia imediatamente ao intervalo de tempo atual do thread, entregando voluntariamente a CPU a outros threads. O novo método Thread.Yield() do Framework 4.0 faz a mesma coisa - exceto que ele renuncia somente aos threads em execução no mesmo processador.

O modo de Thread.Sleep(0) ou Thread.Yield() é ocasionalmente útil no código de produção para ajustes avançados de desempenho. Também é uma excelente ferramenta de diagnóstico para ajudar a descobrir problemas de segurança de threads: se inserir Thread.Yield() em qualquer lugar do seu código cria ou interrompe o programa, você quase certamente tem um erro.

**CONSTRUÇÕES DE BLOQUEIO**

**Lock (objeto)**

O Lock é uma palavra-chave em C#; impede que um thread execute o mesmo bloco de código que outro thread está executando. Esse bloco de código é chamado de código bloqueado. Portanto, se um thread tentar inserir um código bloqueado, ele aguardará até que o objeto seja liberado. A palavra-chave lock chama Enter no início do bloco e Exit no final do bloco. A melhor prática é usar a palavra-chave lock com um objeto particular ou com uma variável de objeto estático particular para proteger dados comuns a todas as instâncias.

Ao aplicar um Lock o compilador traduz em uma chamada para System.Thread.Monitor. O exemplo abaixo mostra o uso do operador de Lock para corrigir o exemplo anterior.

int num = 0;

int length = 500000;

object \_lock = new object();

//Run on separate thread of threadpool

Task tsk = Task.Run(() =>

{

for (int i = 0; i < length; i++)

{

lock (\_lock)

num = num + 1;

}

});

//Run on Main Thread

for (int i = 0; i < length; i++)

{

lock (\_lock)

num = num - 1;

}

tsk.Wait();

Console.WriteLine(num); //0

Onde:

* lock (\_lock){...}: Impede que outros threads manipulem a memória compartilhada, ou seja, "n". Quando o controle sai do bloco, a memória compartilhada se torna utilizável para qualquer thread.
* \_lock: é a mesma variável usada em vários threads, notificando outros threads se alguém já o usou para bloquear um bloco de código.
* Portanto, a memória compartilhada se torna segura para threads e o programa fornece um resultado preciso

Após essa alteração, o programa sempre gera 0 porque o acesso à variável n agora está sincronizado. Não há como um thread alterar o valor enquanto o outro thread estiver trabalhando com ele. No entanto, também faz com que os threads sejam bloqueados enquanto eles aguardam um pelo outro. Isso pode causar problemas de desempenho e pode até levar a um impasse (deadlock).

**Monitor (métodos Enter/Exit)**

A classe Monitor também garante que nenhum outro thread possa executar a mesma seção de código ou uma memória compartilhada até que seja executado pelo proprietário do bloqueio. Monitores são primitivas de sincronização usadas para sincronizar o acesso a objetos. Eles são implementados no .NET na classe System.Threading.Monitor. A classe Monitor é usada em conjunto com **tipos de referência**, não tipos de valor, para garantir que apenas um thread possa acessar esse objeto por vez.

A razão para não se adotar objetos de valor é que eles são copiados (boxed) quando enviados como parâmetros. O resultado será que o bloqueio será adquirido em um objeto, mas quando você o liberar, estará em outro objeto. Na prática, isso gera um bloqueio completamente novo a cada vez, perdendo o mecanismo de bloqueio. Assim, se você chama-se um Exit em um objeto que você nunca chamou Enter, o .NET lançaria uma exceção. Felizmente, o compilador ajuda a gerar um erro quando você acidentalmente usa um tipo de valor para a instrução lock.

A melhor prática para lidar com bloqueios é criando um objeto de referência que é privado para a classe e que será usado apenas para esse fim. Um objeto público pode ser usado por outros threads para adquirir um bloqueio sem que seu código saiba. Se você precisar lidar com código legado ou procurar amostras na Internet, poderá encontrar um código que esteja bloqueado utilzando o “this” como objeto de referência. O código fica assim:

lock (this) { // Code updating some shared data}

Embora esse seja um snippet de código C# perfeitamente válido e, de fato, adquire e libera um bloqueio no objeto atual, esse código possui um erro latente que pode se manifestar a qualquer momento. Isso porque a variável “this” pode ser usada por outro código para criar um bloqueio, causando conflitos. Veja o seguinte trecho de código:

public class LockThisBadSample

{

public void OneMethod()

{

lock (this)

{

Console.WriteLine("thislock of OneMethod");

Thread. (100);

}

}

}

public class UsingTheLockedObject

{

public void AnotherMethod()

{

LockThisBadSample lockObject = new LockThisBadSample();

{

// Do something else } }}

}

}

}

Como você pode ver, AnotherMethod adquire um bloqueio no lockObject, que é essa referência dentro da chamada OneMethod. Como você não tem controle sobre todo o código que gostaria de adquirir bloqueios nos objetos, isso pode facilmente levar a conflitos. A lição aqui é evitar o lock (this), mesmo que você veja exemplos online usando esse tipo de programação, e o MSDN não é uma exceção.

Pelo mesmo motivo, você não deve bloquear uma string. Devido à internação de seqüências de caracteres (o processo no qual o compilador cria um objeto para várias seqüências de caracteres que possuem o mesmo conteúdo), você pode estar subitamente solicitando um bloqueio em um objeto usado em vários locais.

Em geral, evite bloquear um tipo público ou instâncias fora do controle do seu código. As construções comuns lock (this), lock (typeof (MyType)) e lock ("myLock") violam esta diretriz:

* lock (this) é um problema se a instância puder ser acessada publicamente.
* lock (typeof (MyType)) é um problema se o MyType estiver acessível ao público.
* lock (“myLock”) é um problema porque qualquer outro código no processo que use a mesma sequência compartilhará o mesmo bloqueio.

A classe expõe apenas métodos estáticos que tomam como primeiro parâmetro o objeto no qual você deseja bloquear. A qualquer momento, no máximo, um thread pode colocar um bloqueio (lock) em um objeto chamando o método estático Monitor.Enter. Se outro thread chamar o Monitor.Enter antes do primeiro thread chamado Monitor.Exit, esse segundo thread será bloqueado até o primeiro thread chamar Monitor.Exit. No .NET, todos os objetos têm um campo que contém uma referência ao thread que adquiriu um bloqueio no objeto, uma lista pronta com todos os threads que desejam adquirir o bloqueio e uma lista de espera com todos os threads aguardando o objeto obter uma notificação através dos métodos Pulse ou PulseAll. A classe expõe vários métodos estáticos, alguns dos quais estão listados na tabela abaixo

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| Enter | Adquire um bloqueio exclusivo em um objeto especificado. Se o bloqueio já foi adquirido por outro thread, o thread atual será colocado na fila de espera e bloqueará sua execução até que o thread que possui o objeto libere o bloqueio. |
| Exit | Libera um bloqueio exclusivo no objeto especificado. |
| IsEntered | Retorna true se o thread atual mantém o bloqueio no objeto especificado. Este método foi introduzido no .NET 4.5. |
| TryEnter | Tenta adquirir um bloqueio exclusivo no objeto especificado. Este método possui seis sobrecargas, permitindo que você especifique um tempo limite também. |

A linguagem C # fornece uma instrução lock como um atalho para o Monitor. O exemplo anterior poderia ter sido usado o Monitor da seguinte forma:

object \_lock = new object();

for (int i = 0; i < length; i++)

{

//lock the block of code

Monitor.Enter(\_lock);

try

{

num = num + 1;

}

finally

{

//unlock the locked code

Monitor.Exit(\_lock);

}

}

Onde:

* O método Monitor.Enter ou Monitor.TryEnter é usado para bloquear um bloco de código para outros threads e impedir a execução de outros threads.
* O método Monitor.Exit é usado para desbloquear o código bloqueado para outro thread e permitir que outros threads o executem.
* Ao utilizar o try/catch, você garante que, mesmo que o código gere uma exceção, o bloqueio ainda seja liberado, resolvendo prblemas de deadlock.

**Mutex**

As duas principais construções exclusivas de bloqueio são lock/Monitores e Mutex. Dos dois, a construção de lock/Monitores é mais rápida e mais conveniente, mas garantem a segurança apenas dos threads que é gerada por um aplicativo e não tem controle sobre os threads provenientes de fora de um aplicativo. O Mutex, no entanto, tem um nicho em que seu bloqueio pode abranger aplicativos em diferentes processos no computador. Em outras palavras, o Mutex pode ser de todo o computador e de todo o aplicativo. Adquirir e liberar um Mutex sem assistência leva alguns microssegundos - cerca de 50 vezes mais lento que um lock.

Com uma classe Mutex, você chama o método WaitOne para bloquear e ReleaseMutex para desbloquear. Fechar ou descartar um Mutex o libera automaticamente. Assim como na instrução lock, um Mutex pode ser liberado apenas a partir do mesmo thread que o obteve. Um uso comum para um Mutex entre processos é garantir que apenas uma instância de um programa possa ser executada por vez. Aqui está como é feito:

static Mutex m1 = new Mutex(true, "Questpond");

static void Main(string[] args)

{

if (IsInstance() == true)

{

Console.WriteLine("New Instance created...");

}

else

{

Console.WriteLine("Instance already acquired...");

}

Console.ReadLine();

}

static bool IsInstance()

{

if (m1.WaitOne(5000, false) == false)

return false;

else

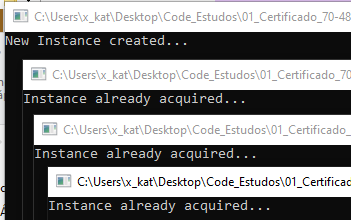
return true;

}

No código acima, criamos uma função booleana chamada "IsInstance", que verifica se alguma outra instância está em execução ou não, e que alcançamos o uso da função booleana "WaitOne" que bloqueia a atual unidade de thread que o manipulador recebe o sinal. Aqui em nossa demonstração, o WaitOne espera por 5 segundos para receber o sinal do cabo de espera, se nenhum sinal for recebido até 5 segundos, ele retornará automaticamente falso. Se for um novo thread, ele retornará automaticamente true. Para ver o cenário em tempo real, siga estas etapas.

1. Crie o código fonte completo
2. Ir paraa pasta BIN do aplicativo
3. Clique no exe e crie threads externos.
4. Não feche o exe anterior e abra mais 2 ou 3 exe.

A primeira linha de comando aberta exibirá a mensagem como: Nova instância criada Todas as outras linhas de comando abertas exibirão mensagens como: Instância já adquirida



**Estrutura SpinLock**

A estrutura System.Threading.SpinLock, como Monitor, concede acesso exclusivo a um recurso compartilhado com base na disponibilidade de um bloqueio. Quando SpinLock tenta adquirir um bloqueio que não está disponível, ele aguarda em um loop, verificando repetidamente até o bloqueio ficar disponível.

Usar um SpinLock é como usar um lock/Monitor comum, exceto:

* Spinlocks são estruturas.
* Spinlocks não são reentrantes, o que significa que você não pode chamar Enter no mesmo SpinLock duas vezes seguidas no mesmo thread. Se você violar essa regra, ela lançará uma exceção (se o rastreamento do proprietário estiver ativado) ou um deadlock (se o rastreamento do proprietário estiver desativado). Você pode especificar se deseja ativar o rastreamento do proprietário ao construir o SpinLock. O rastreamento do proprietário gera um impacto no desempenho.
* O SpinLock permite que você pergunte se o bloqueio foi realizado, através das propriedades IsHeld e, se o rastreamento do proprietário estiver ativado, IsHeldByCurrentThread.

Outra diferença é que, quando você chama Enter, deve seguir o padrão robusto de fornecer um argumento gotLock (que quase sempre é feito dentro de um bloco try/finally). Aqui está um exemplo:

SpinLock sl = new SpinLock();

StringBuilder sb = new StringBuilder();

// Action taken by each parallel job.

// Append to the StringBuilder 10000 times, protecting

// access to sb with a SpinLock.

Action action = () =>

{

bool gotLock = false;

for (int i = 0; i < 10000; i++)

{

gotLock = false;

try

{

sl.Enter(ref gotLock);

sb.Append((i % 10).ToString());

}

finally

{

// Only give up the lock if you actually acquired it

if (gotLock) sl.Exit();

}

}

};

// Invoke 3 concurrent instances of the action above

Parallel.Invoke(action, action, action);

// Check/Show the results

Console.WriteLine("sb.Length = {0} (should be 30000)", sb.Length);

Console.WriteLine("number of occurrences of '5' in sb: {0} (should be 3000)",

sb.ToString().Where(c => (c == '5')).Count());

Assim como ocorre com um bloqueio comum, o gotLock será falso após chamar Enter se (e somente se) o método Enter lançar uma exceção e o bloqueio não tiver sido realizado. Isso acontece em cenários muito raros (como Abort ser chamado no thread ou saída de OutOfMemoryException) e permite saber com segurança se deve chamar Exit posteriormente. O SpinLock também fornece um método TryEnter que aceita um tempo limite.

Um SpinLock faz mais sentido ao escrever suas próprias construções de sincronização reutilizáveis. Mesmo assim, um spinlock não é tão útil quanto parece. Ainda limita a simultaneidade. E desperdiça tempo de CPU sem fazer nada útil. Freqüentemente, uma escolha melhor é passar parte desse tempo fazendo algo especulativo - com a ajuda do SpinWait.

**Estrutura SpinWait**

A estrutura System.Threading.SpinWait oferece suporte para espera baseada em rotação. Você pode usá-la quando um thread tiver de esperar pela sinalização de um evento ou por uma condição específica. No entanto, quando o tempo de espera real for menor do que o tempo necessário, use um identificador de espera ou bloqueie o thread. Usando o SpinWait, você pode especificar um curto período de tempo para girar enquanto espera e, em seguida, gerar (por exemplo, aguardando ou em espera) somente se a condição não for atendida no tempo especificado.

bool someBoolean = false;

int numYields = 0;

// First task: SpinWait until someBoolean is set to true

Task t1 = Task.Factory.StartNew(() =>

{

SpinWait sw = new SpinWait();

while (!someBoolean)

{

// The NextSpinWillYield property returns true if

// calling sw.SpinOnce() will result in yielding the

// processor instead of simply spinning.

if (sw.NextSpinWillYield) numYields++;

sw.SpinOnce();

}

// Az .NET Framework 4: After some initial spinning, SpinWait.SpinOnce() will yield every time.

Console.WriteLine("SpinWait called {0} times, yielded {1} times", sw.Count, numYields);

});

// Second task: Wait 100ms, then set someBoolean to true

Task t2 = Task.Factory.StartNew(() =>

{

Thread.Sleep(100);

someBoolean = true;

});

// Wait for tasks to complete

Task.WaitAll(t1, t2);

**Semaphore**

Um semáforo é como uma boate: tem uma certa capacidade, imposta por um segurança. Quando estiver cheio, não haverá mais pessoas entrando e uma fila se acumulará do lado de fora. Em seguida, para cada pessoa que sai, uma pessoa entra no início da fila. O construtor exige um mínimo de dois argumentos: o número de vagas atualmente disponíveis na boate e a capacidade total do clube.

Um semáforo com capacidade de um é semelhante a um Mutex ou lock, exceto que o semáforo não tem "proprietário" - ele é independente de thread. Qualquer thread pode chamar Release em um semáforo, enquanto que com o Mutex e o lock, somente o thread que obteve o bloqueio pode liberá-lo.

Um semáforo pode ser chamado de uma versão avançada do Mutex com recursos adicionais. O semáforo também nos ajuda a trabalhar com threads externos e identificar se um aplicativo é adquirido por um thread externo ou não. Mas, diferentemente do Mutex, o Semaphore permite que um ou mais threads entrem para executar sua tarefa com segurança de thread. Melhor característica do semáforo que podemos limitar o número de threads a inserir.

static Semaphore s1 = new Semaphore(2, 2, "SemaphoreQuestpond");

static void Main(string[] args)

{

if (IsInstance() == true)

{

Console.WriteLine("New Instance created...");

}

else

{

Console.WriteLine("Instance already acquired...");

}

Console.ReadLine();

}

static bool IsInstance()

{

if (s1.WaitOne(5000, false) == false)

return false;

else

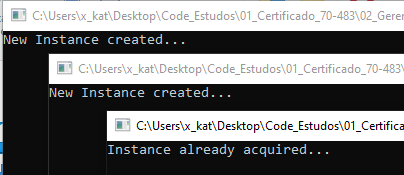
return true;

}

Como você viu no código acima, criamos um limite de 2 threads para o objeto semáforo "s1" com o nome "SemaphoreQuestpond". Agora examinaremos o exemplo acima se o semáforo nos permite passar dois threads externos ao mesmo tempo, mantendo a segurança do thread.

1. Primeiro crie o código fonte completo
2. Vá para a pasta bin e clique no arquivo exe duas vezes
3. Sem fechar o arquivo exe do arquivo de comando, clique no arquivo exe novamente

As duas primeiras linhas de comando abertas exibirão a mensagem como: Nova instância criada Todas as outras linhas de comando abertas exibirão mensagens como: Instância já adquirida.



**SemaphoreSlim**

Esta é ujma versão da classe Semaphore (introduzida no Framework 4.0 ) que foi otimizado para atender às demandas de baixa latência da programação paralela. Também é útil no multithreading tradicional, pois permite especificar um token de cancelamento ao aguardar. No entanto, ele não pode ser usado para sinalização interprocessos. O semáforo incorre em aproximadamente 1 microssegundo na chamada WaitOne ou Release; O SemaphoreSlim incorre em cerca de um quarto disso.

SemaphoreSlim é uma versão avançada do Monitor. O SemaphoreSlim garante a segurança do thread com threads internos mas permite-nos passar um ou mais threads para executar sua tarefa. O SemaphoreSlim também fornece um limite avançado, no qual você pode limitar o número de threads para uma execução.

Os semáforos podem ser úteis para limitar a simultaneidade, impedindo que muitos threads executem uma parte específica do código de uma só vez. No exemplo a seguir, cinco threads tentam entrar em uma boate que permite apenas três threads ao mesmo tempo:

static SemaphoreSlim \_sem = new SemaphoreSlim(3); // Capacity of 3

static void Main(string[] args)

{

for (int i = 1; i <= 5; i++) new Thread(Enter).Start(i);

Console.ReadKey();

}

static void Enter(object id)

{

Console.WriteLine(id + " wants to enter");

\_sem.Wait();

Console.WriteLine(id + " is in!"); // Only three threads

Thread.Sleep(1000 \* (int)id); // can be here at

Console.WriteLine(id + " is leaving"); // a time.

\_sem.Release();

}

Se a instrução Sleep estivesse executando E/S de disco intensivas, o SemaphoreSlim melhoraria o desempenho geral, limitando a atividade simultânea excessiva do disco rígido. Um SemaphoreSlim, se nomeado, pode abranger processos da mesma maneira que um Mutex.

**Classe ReaderWriterLockSlim**

O problema com a classe efetivamente descontinuada ReaderWriterLock está na sua implementação. Vários especialistas criticaram essa técnica e descobriram que, fora de cenários limitados, na verdade, é muito mais lenta que o método Monitor.Enter usado para obter um bloqueio exclusivo. O ReaderWriterLock dá maior prioridade aos threads do leitor do que os gravadores. Assim em cenários que tiver muitos leitores e apenas alguns escritores faz sentido utilizar o ReaderWriterLock. Mas e se você tiver escritores iguais ou mais, o processo de favorecer os leitores faz com que os threads dos escritores enfileirem e demorem muito tempo para serem concluídos.

Para resolver esses problemas relacionados à classe ReaderWriterLock, o .Net Framework 3.5 introduziu a classe ReaderWriterLockSlim. A classe System.Threading.ReaderWriterLockSlim concede acesso exclusivo a um recurso compartilhado para gravação e permite que vários threads acessem o recurso simultaneamente para leitura. Você talvez queira usar ReaderWriterLockSlim para sincronizar o acesso a uma estrutura de dados compartilhada que dá suporte a operações de leitura thread-safe, mas requer acesso exclusivo para realizar a operação de gravação. Quando um thread solicita acesso exclusivo (por exemplo, ao chamar o método ReaderWriterLockSlim.EnterWriteLock), o leitor subsequente solicita o bloqueio até que todos os leitores e gravadores existentes tenham saído do bloqueio e o gravador tenha entrado e saído do bloqueio.

No código a seguir, três threads continuamente enumeram uma lista, enquanto outros dois threads acrescentam um número aleatório à lista a cada segundo. Um bloqueio de leitura protege os leitores da lista e um bloqueio de gravação protege os gravadores da lista:

static ReaderWriterLockSlim rw = new ReaderWriterLockSlim();

static List<int> items = new List<int>();

static Random rand = new Random();

static void Main(string[] args)

{

new Thread(Read).Start();

new Thread(Read).Start();

new Thread(Read).Start();

new Thread(Write).Start("A");

new Thread(Write).Start("B");

Console.Read();

}

static void Read()

{

while (true)

{

rw.EnterReadLock();

foreach (int i in items) Thread.Sleep(10);

rw.ExitReadLock();

}

}

static void Write(object threadID)

{

while (true)

{

int newNumber = GetRandNum(50);

rw.EnterWriteLock();

items.Add(newNumber);

rw.ExitWriteLock();

Console.WriteLine("Thread " + threadID + " added " + newNumber);

Thread.Sleep(100);

}

}

static int GetRandNum(int max)

{

lock (rand)

return rand.Next(max);

}

Quando comparado a um bloqueio comum (Monitor.Enter/Exit), o ReaderWriterLockSlim é duas vezes mais lento.

**CONSTRUÇÕES DE SINALIZAÇÃO**

Sinalização é quando um thread aguarda até receber notificação de outro.

**Classes EventWaitHandle**

A classe System.Threading.EventWaitHandle representa um evento de sincronização de thread. Um evento de sincronização pode estar em um estado não sinalizado ou sinalizado. Quando o estado de um evento não é sinalizado, um thread que chama a sobrecarga WaitOne do evento é bloqueado até que um evento seja sinalizado. O método EventWaitHandle.Set define o estado de um evento como sinalizado. EventWaitHandle é definido no namespace System.Threading e a tabela abaixo lista os métodos mais comuns.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| EventWaitHandle | Construtor. Este método possui quatro sobrecargas diferentes. No mínimo, você precisa especificar se o evento deve ser sinalizado e se o evento deve ser redefinido manual ou automaticamente usando a enumeração EventResetMode. |
| Dispose | Este é o método da interface IDisposable. Você precisa chamar esse método para garantir que os recursos do SO sejam liberados quando esse objeto não for mais necessário. |
| Reset | Define o estado do evento para um estado sem sinal, causando o bloqueio de threads. |
| Set | Define o estado do evento para o estado sinalizado. Um ou mais threads em espera poderão continuar. Se o evento foi criado como redefinição automática, apenas um thread será ativado para chamar WaitOne sem ser bloqueado. Ou se houver threads já bloqueados como resultado de uma chamada para WaitOne, apenas um thread será desbloqueado e o evento será novamente sem sinal até que o método Set seja chamado novamente. Se o evento foi criado como ManualReset, o evento será sinalizado até que Reset seja chamado nesse evento. |
| WaitOne | Bloqueia o thread atual se o evento não tiver sinal. Quando esse evento é sinalizado, se ele foi criado como redefinição automática, ele desbloqueia o thread e redefine o evento no estado não sinalizado. |

No Windows, você pode usar EventWaitHandle para a sincronização entre processos. Para fazer isso, crie uma instância EventWaitHandle que representa um evento de sincronização do sistema nomeado usando um dos construtores EventWaitHandle que especificam um nome ou o método EventWaitHandle.OpenExisting.

Aqui está um exemplo de uso desta classe. E, para ser mais preciso, é possível ver como corrigir a solução do conjunto de threads para garantir que você não esteja tentando ler o resultado do cálculo antes que o cálculo seja realmente concluído. O método original seria assim:

static double ReadDataFromIO()

{

// We are simulating an I/O by putting the current thread to sleep.

Thread.Sleep(5000);

return 10d;

}

static double DoIntensiveCalculations()

{

// We are simulating intensive calculations

// by doing nonsens divisions

double result = 100000000d;

var maxValue = Int32.MaxValue;

for (int i = 1; i < maxValue; i++)

{

result /= i;

}

return result + 10d;

}

static void RunInThreadPool()

{

double result = 0d;

// Create a work item to read from I/O

ThreadPool.QueueUserWorkItem((x) => result += ReadDataFromIO());

// Save the result of the calculation into another variable

double result2 = DoIntensiveCalculations();

// Calculate the end result

result += result2;

// Print the result

Console.WriteLine("The result is {0}", result);

}

Uma possível solução usando sinalização é semelhante a esta:

static void RunInThreadPoolWithEvents()

{

double result = 0d;

// We use this event to signal when the thread is don executing.

EventWaitHandle calculationDone =

new EventWaitHandle(false, EventResetMode.ManualReset);

// Create a work item to read from I/O

ThreadPool.QueueUserWorkItem((x) => {

result += ReadDataFromIO();

calculationDone.Set();

});

// Save the result of the calculation into another variable

double result2 = DoIntensiveCalculations();

// Wait for the thread to finish

calculationDone.WaitOne();

// Calculate the end result

result += result2;

// Print the result

Console.WriteLine("The result is {0}", result);

}

O código anterior faz o seguinte:

1. O código primeiro cria um objeto EventWaitHandle no estado não sinalizado.
2. O código enfileira um novo item de trabalho. Depois de obter o primeiro resultado, você sinaliza o evento para indicar que o cálculo foi feito.
3. No thread principal, você chama o segundo método.
4. Após o retorno do segundo método, é necessário aguardar o primeiro cálculo, aguardando a sinalização do evento.
5. Quando você recebe o sinal, sabe que possui o resultado, para que possa calcular o resultado final e mostrá-lo.

**Criando um EventWaitHandle de processo cruzado**

O construtor EventWaitHandle permite que um EventWaitHandle "nomeado" seja criado, capaz de operar em vários processos. O nome é simplesmente uma string e pode ser qualquer valor que não entre em conflito acidentalmente com o de outra pessoa! Se o nome já estiver em uso no computador, você obterá uma referência ao mesmo EventWaitHandle subjacente; caso contrário, o sistema operacional cria um novo. Aqui está um exemplo:

EventWaitHandle wh = new EventWaitHandle(false, EventResetMode.AutoReset,"MyCompany.MyApp.SomeName");

Se dois aplicativos executassem esse código, eles poderiam sinalizar um ao outro: o identificador de espera funcionaria em todos os threads nos dois processos.

**AutoResetEvent, ManualResetEvent e ManualResetEventSlim**

O .NET fornece duas classes que herdam de EventWaitHandle: AutoResetEvent e ManualResetEvent. Ambas as classes têm apenas um construtor e nenhum método ou propriedade própria definida. Nos dois casos, o construtor utiliza um parâmetro booleano que especifica se o evento é sinalizado inicialmente. O comportamento de um EventWaitHandle que foi assinalado depende de seu modo de redefinição:

* **EventResetMode.AutoReset:** com este sinalizador o EventWaitHandle é redefinido, volta automaticamente para sem sinal, após liberar um único thread em espera (ser sinalizado) , exatamente como o método Monitor.Pulse. É como uma roleta/catraca/pedágio que permite que apenas um thread de cada vez seja sinalizado.
* **EventResetMode.ManualReset:** com este sinalizador o EventWaitHandle permanece sinalizado até que seu método Reset seja chamado. É como um portão que é fechado até ser sinalizado e então permanece aberto até que alguém o feche. ManualResetEvent pode ser descrito como uma mangueira de água, uma vez aberta, ela deixa tudo passar até que você a feche.
* **System.Threading.ManualResetEventSlim:** é uma alternativa leve para ManualResetEvent.

**AutoResetEvent**

Um AutoResetEvent é como uma catraca de ticket: a inserção de um ticket permite que exatamente uma pessoa passe. O "Auto" no nome da classe refere-se ao fato de que uma catraca aberta é fechada automaticamente ou "redefinida" depois que alguém avança, um tíquete é inserido chamando o método Set. Quando um thread atinge WaitOne(),bloqueia a catraca, e espera até que outras chamadas de thread sejam definidas(). O AutoResetEvent (catraca) é redefinido automaticamente quando o thread em espera observa que o evento é sinalizado (definido).

Além de conveniente, se você estiver reutilizando o evento várias vezes, ele tem uma aplicação prática: se houver vários threads aguardando um evento de redefinição automática, apenas um deles será ativado quando o evento for definido. Se um número de threads chamar WaitOne, uma fila será construída na catraca. (Como ocorre com os bloqueios, a justiça da fila às vezes pode ser violada devido a nuances no sistema operacional). Um ticket pode vir de qualquer thread; em outras palavras, qualquer thread (desbloqueado) com acesso ao objeto AutoResetEvent pode chamar Set para liberar um thread bloqueado.

O AutoResetEvent permite que os threads se comuniquem entre si sinalizando. Normalmente, essa comunicação diz respeito a um recurso ao qual os threads precisam de acesso exclusivo. Você pode criar um AutoResetEvent de duas maneiras. O primeiro é via seu construtor:

var auto = new AutoResetEvent(false);

Podemos controlar o estado inicial de um AutoResetEvent inserindo um valor booleano no construtor. Passar true para o construtor é equivalente a chamar imediatamente Set nele. A segunda maneira de criar um AutoResetEvent é a seguinte:

var auto = new EventWaitHandle(false, EventResetMode.AutoReset);

O AutoResetEvent também pode ser usado com o método estático: WaitAll () e WaitAny (). A seguir um aplicativo de console simples para demonstrar esta classe.

public class Process

{

AutoResetEvent auto;

public Process()

{

auto = new AutoResetEvent(false);

Thread t1 = new Thread(new ThreadStart(akshay));

Thread t2 = new Thread(new ThreadStart(csharpcorner));

t1.Start();

t2.Start();

auto.Set();

Thread.Sleep(1000);

auto.Set();

}

void akshay()

{

auto.WaitOne();

for (int i = 0; i < 10; i++)

{

Thread.Sleep(500);

Console.WriteLine("Akshay Teotia");

}

}

void csharpcorner()

{

auto.WaitOne();

for (int i = 0; i < 10; i++)

{

Thread.Sleep(500);

Console.WriteLine("csharpcorner.com");

}

}

}

Se Set for chamado quando nenhum thread estiver aguardando, o identificador permanecerá aberto pelo tempo necessário até que algum thread chame WaitOne. Esse comportamento ajuda a evitar uma condição de corrida entre um cabeçalho da catraca e um thread que insere um ticket. No entanto, chamar Set repetidamente em uma catraca na qual ninguém está esperando não permite que todos possam entrar de uma vez: apenas uma pessoa seguinte é liberada e os ingressos extras são "desperdiçados".

A redefinição de chamada em um AutoResetEvent fecha a catraca (caso esteja aberta) sem esperar ou bloquear.

WaitOne aceita um parâmetro opcional de tempo limite, retornando false se a espera terminar devido a um tempo limite em vez de obter o sinal. Chamar o WaitOne com um tempo limite de 0 testa se um identificador de espera está "aberto", sem bloquear o chamador. Porém, lembre-se de que isso redefine o AutoResetEvent, se estiver aberto.

**Sinalização AutoResetEvent bidirecional**

Digamos que queremos que o thread principal sinalize um thread de trabalho três vezes seguidas. Se o thread principal simplesmente chamar Conjunto em uma alça de espera várias vezes em rápida sucessão, o segundo ou terceiro sinal poderá se perder, pois o trabalhador pode demorar para processar cada sinal.

A solução é que o thread principal aguarde até que o trabalhador esteja pronto antes de sinalizá-lo. Isso pode ser feito com outro AutoResetEvent, da seguinte maneira:

static EventWaitHandle \_ready = new AutoResetEvent(false);

static EventWaitHandle \_go = new AutoResetEvent(false);

static readonly object \_locker = new object();

static string \_message;

static void Main(string[] args)

{

new Thread(Work).Start();

\_ready.WaitOne(); // First wait until worker is ready

lock (\_locker) \_message = "ooo";

\_go.Set(); // Tell worker to go

\_ready.WaitOne();

lock (\_locker) \_message = "ahhh"; // Give the worker another message

\_go.Set();

\_ready.WaitOne();

lock (\_locker) \_message = null; // Signal the worker to exit

\_go.Set();

Console.ReadKey();

}

static void Work()

{

while (true)

{

\_ready.Set(); // Indicate that we're ready

\_go.WaitOne(); // Wait to be kicked off...

lock (\_locker)

{

if (\_message == null) return; // Gracefully exit

Console.WriteLine(\_message);

}

}

}

Aqui, estamos usando uma mensagem nula para indicar que o trabalhador deve terminar. Com threads que funcionam indefinidamente, é importante ter uma estratégia de saída!

**Fila produtor/consumidor AutoResetEvent**

Uma fila de produtor / consumidor é um requisito comum no thread. Eis como funciona:

* Uma fila é configurada para descrever itens de trabalho - ou dados nos quais o trabalho é executado.
* Quando uma tarefa precisa ser executada, ela é colocada na fila, permitindo que o chamador continue com outras coisas.
* Um ou mais threads de trabalho são conectados em segundo plano, selecionando e executando itens na fila.

A vantagem desse modelo é que você tem um controle preciso sobre quantos threads de trabalho executam ao mesmo tempo. Isso permite limitar o consumo não apenas do tempo da CPU, mas também de outros recursos. Se as tarefas executam E / S de disco intensivas, por exemplo, você pode ter apenas um thread de trabalho para evitar a fome do sistema operacional e de outros aplicativos. Outro tipo de aplicativo pode ter 20. Você também pode adicionar e remover trabalhadores dinamicamente ao longo da vida da fila. O próprio conjunto de threads do CLR é um tipo de fila de produtor / consumidor.

Uma fila de produtor/consumidor normalmente mantém itens de dados nos quais (a mesma) tarefa é executada. Por exemplo, os itens de dados podem ser nomes de arquivos e a tarefa pode ser criptografar esses arquivos.

No exemplo abaixo, usamos um único AutoResetEvent para sinalizar um trabalhador, que aguarda quando a tarefa fica sem tarefas (em outras palavras, quando a fila está vazia). Encerramos o trabalhador enfileirando uma tarefa nula:

class ProducerConsumerQueue : IDisposable

{

EventWaitHandle \_wh = new AutoResetEvent(false);

Thread \_worker;

readonly object \_locker = new object();

Queue<string> \_tasks = new Queue<string>();

public ProducerConsumerQueue()

{

\_worker = new Thread(Work);

\_worker.Start();

}

public void EnqueueTask(string task)

{

lock (\_locker) \_tasks.Enqueue(task);

\_wh.Set();

}

public void Dispose()

{

EnqueueTask(null); // Signal the consumer to exit.

\_worker.Join(); // Wait for the consumer's thread to finish.

\_wh.Close(); // Release any OS resources.

}

void Work()

{

while (true)

{

string task = null;

lock (\_locker)

if (\_tasks.Count > 0)

{

task = \_tasks.Dequeue();

if (task == null) return;

}

if (task != null)

{

Console.WriteLine("Performing task: " + task);

Thread.Sleep(1000); // simulate work...

}

else \_wh.WaitOne(); // No more tasks - wait for a signal

}

}

}

Para garantir a segurança do thread, usamos um bloqueio para proteger o acesso à coleção Fila <string>. Também encerramos explicitamente a alça de espera em nosso método Dispose, pois poderíamos criar e destruir muitas instâncias dessa classe na vida útil do aplicativo. Aqui está um método principal para testar a fila:

using (ProducerConsumerQueue q = new ProducerConsumerQueue())

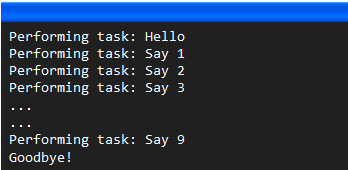
{

q.EnqueueTask("Hello");

for (int i = 0; i < 10; i++) q.EnqueueTask("Say " + i);

q.EnqueueTask("Goodbye!");

}



Depois de terminar com uma WaitHandles, você pode chamar o método Close para liberar o recurso do sistema operacional. Como alternativa, você pode simplesmente eliminar todas as referências à WaitHandles e permitir que o coletor de lixo faça o trabalho para você algum tempo depois (as WaitHandles implementam o padrão de descarte pelo qual o finalizador chama Close). Esse é um dos poucos cenários em que a confiança nesse backup é aceitável, porque os identificadores de espera têm uma carga leve do sistema operacional (delegados assíncronos contam exatamente com esse mecanismo para liberar o identificador de espera do IAsyncResult). Os identificadores de espera são liberados automaticamente quando um domínio de aplicativo é descarregado.

O Framework 4.0 fornece uma nova classe chamada BlockingCollection <T> que implementa a funcionalidade de uma fila de produtores / consumidores. Nossa fila de produtor/consumidor gravada manualmente ainda é valiosa - não apenas para ilustrar o AutoResetEvent e a segurança de threads, mas também como base para estruturas mais sofisticadas. Por exemplo, se desejássemos uma fila de bloqueio limitada (limitando o número de tarefas na fila) e também desejássemos oferecer suporte ao cancelamento (e remoção) de itens de trabalho na fila, nosso código forneceria um excelente ponto de partida.

**ManualResetEvent**

ManualResetEvent permite que os threads se comuniquem entre si através de sinalização. Normalmente, essa comunicação diz respeito a uma tarefa que um thread deve concluir antes que outros threads possam prosseguir. É como um portão que permite mais de um de cada vez. Quando um thread atinge WaitOne (), ele espera até que outro thread chame Set ().

Um ManualResetEvent é útil se você deseja ativar um monte de threads com um único evento. É útil para coisas como eventos de encerramento em que você não deseja redefinir o evento apenas porque observou o sinal. Se o significado do evento for mais parecido com um sinalizador, você poderá observá-lo sem alterar seu estado novamente para não sinalizado.

Quando um thread inicia uma atividade que deve ser concluída antes que outros threads prossigam, ele chama Reset para colocar ManualResetEvent no estado não sinalizado. Esse thread pode ser considerado como o controle do ManualResetEvent. Os threads que chamam WaitOne () no ManualResetEvent serão bloqueados, aguardando o sinal. Quando o thread de controle conclui a atividade, chama Set para sinalizar que os threads em espera podem prosseguir. Todos os threads em espera são liberados.

Depois de sinalizado, ManualResetEvent permanece sinalizado até que seja redefinido manualmente. Ou seja, as chamadas para WaitOne () retornam imediatamente. Como AutoResetEvent, podemos controlar o estado inicial do ManualResetEvent, inserindo um valor booleano no construtor, true se o estado inicial for sinalizado e false, caso contrário. ManualResetEvent também pode ser usado com os métodos estáticos: WaitAll () e WaitAny (). A seguir um aplicativo de console simples para demonstrar esta classe.

class MyThread

{

public Thread th;

ManualResetEvent manualResetEvent;

public MyThread(string name, ManualResetEvent e)

{

th = new Thread(this.run);

th.Name = name;

manualResetEvent = e;

th.Start();

}

void run()

{

Console.WriteLine("Inside thread " + th.Name);

for (int i = 0; i < 10; i++)

{

Console.WriteLine(th.Name);

Thread.Sleep(50);

}

Console.WriteLine(th.Name + " Done!");

manualResetEvent.Set();

}

}

ManualResetEvent evtObj = new ManualResetEvent(false);

MyThread myThread = new MyThread("Event Thread 1", evtObj);

Console.WriteLine("Main thread waiting for event.");

// Wait for signaled event.

evtObj.WaitOne();

Console.WriteLine("Main thread received first event.");

evtObj.Reset();

myThread = new MyThread("Event Thread 2", evtObj);

// Wait for signaled event.

evtObj.WaitOne();

Console.WriteLine("Main thread received second event.");

Console.Read();

**ManualResetEventSlim**

No Framework 4.0, há outra versão do ManualResetEvent chamada ManualResetEventSlim. O último é otimizado para curtos tempos de espera - com a capacidade de ativar a rotação para um número definido de iterações. Ele também possui uma implementação gerenciada mais eficiente e permite que uma espera seja cancelada por meio de um CancellationToken. No entanto, ele não pode ser usado para sinalização interprocessos. ManualResetEventSlim não inclui a subclasse WaitHandle; no entanto, expõe uma propriedade WaitHandle que retorna um objeto baseado em WaitHandle quando chamado (com o perfil de desempenho de um identificador de espera tradicional).

Aguardar ou sinalizar um AutoResetEvent ou ManualResetEvent leva cerca de um microssegundo (presumindo que não haja bloqueio). ManualResetEventSlim e CountdownEvent podem ser até 50 vezes mais rápidos em cenários de espera curta, devido à sua falta de confiança no sistema operacional e ao uso criterioso de construções giratórias (spinning). Na maioria dos cenários, no entanto, a sobrecarga das próprias classes de sinalização não cria um gargalo e, portanto, raramente é uma consideração.

**Wait Handles e o ThreadPool**

Se seu aplicativo tiver muitos threads que passam a maior parte do tempo bloqueados em um identificador de espera, você pode reduzir a carga de recursos chamando ThreadPool.RegisterWaitForSingleObject. Este método aceita um delegado que é executado quando um identificador de espera é sinalizado. Enquanto aguarda, ele não amarra um fio:

static ManualResetEvent \_starter = new ManualResetEvent(false);

static void Main(string[] args)

{

RegisteredWaitHandle reg = ThreadPool.RegisterWaitForSingleObject

(\_starter, Go, "Some Data", -1, true);

Thread.Sleep(5000);

Console.WriteLine("Signaling worker...");

\_starter.Set();

Console.ReadLine();

reg.Unregister(\_starter); // Clean up when we’re done.

}

public static void Go(object data, bool timedOut)

{

Console.WriteLine("Started - " + data);

// Perform task...

}

Quando o identificador de espera é sinalizado (ou o tempo limite se esgota), o delegado é executado em um thread em pool.

Além do identificador de espera e do delegado, o RegisterWaitForSingleObject aceita um objeto de "caixa preta" que ele passa para o método delegado (como ParameterizedThreadStart), além de um tempo limite em milissegundos (–1 significando sem tempo limite) e um sinalizador booleano indicando se a solicitação é pontual e não recorrente.

O RegisterWaitForSingleObject é particularmente valioso em um servidor de aplicativos que deve lidar com muitas solicitações simultâneas. Suponha que você precise bloquear um ManualResetEvent e simplesmente chamar WaitOne:

void AppServerMethod()

{

\_starter.WaitOne();

// ... continue execution

}

Se 100 clientes chamarem esse método, 100 threads do servidor serão vinculados pela duração do bloqueio. Substituir \_wh.WaitOne por RegisterWaitForSingleObject permite que o método retorne imediatamente, sem desperdiçar threads:

void AppServerMethod2()

{

RegisteredWaitHandle reg = ThreadPool.RegisterWaitForSingleObject

(\_starter, Resume, null, -1, true);

}

static void Resume(object data, bool timedOut)

{

// ... continue execution

}

O objeto de dados passado para Continuar permite a continuidade de qualquer dado temporário.

**Monitor (métodos Wait/Pulse/PulseAll)**

A classe de monitor possui dois métodos estáticos Wait and Pulse que fornecem um mecanismo de sinalização simples, resumidamente o Wait bloqueia até receber notificação de outro thread; Pulse fornece essa notificação.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| Pulse | Notifica o thread na fila de espera que o estado do objeto bloqueado foi alterado, movendo o thread da fila de espera para a fila de pronta. |
| PulseAll | Notifica todos os threads em espera que o estado do objeto bloqueado foi alterado, movendo todos os threads da fila de espera para a fila de pronta. |
| Wait | Libera o bloqueio exclusivo no objeto e bloqueia o thread atual até recuperar o bloqueio. O thread atual será colocado na fila de espera e ficará em estado de espera até outro thread chamar Pulse ou PulseAll para que possa retomar sua execução |

A Wait deve ser executada antes do Pulse para que o sinal funcione. Se o Pulse for executado primeiro, seu pulso será perdido e o Wait deverá aguardar um novo Pulse ou permanecer bloqueado para sempre. Isso difere do comportamento de um **AutoResetEvent**, em que seu método Set tem um efeito de "travamento" e, portanto, é eficaz se chamado antes do WaitOne. São as construções mais versáteis, mas são mais difíceis de usar corretamente do que outras primitivas de sincronização como AutoResetEvent.

É necessário especificar um objeto de sincronização, uma instrução lock, ao chamar Wait ou Pulse. Se dois threads usarem o mesmo objeto, eles poderão sinalizar um ao outro. Se eles forem chamados enquanto não estiverem bloqueado antes de chamar Wait ou Pulse, eles lançarão a SynchronizationLockException. Por exemplo:

readonly object key = new object();

// thread A

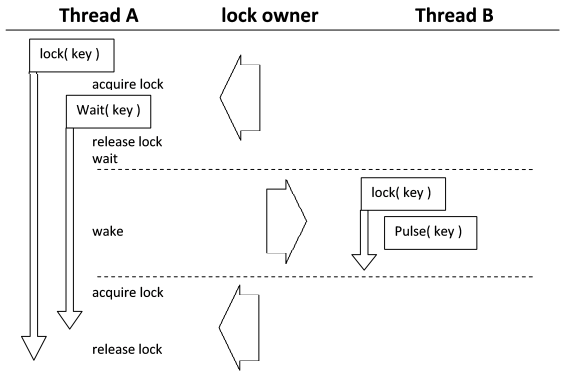
lock (key) Monitor.Wait(key);

// thread B

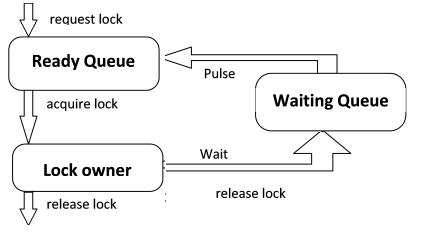
lock (key) Monitor.Pulse(key);

Quando um thread é temporariamente impedido de executar, ele chama Wait() Isso faz com que o thread entre em suspensão e o bloqueio desse objeto seja liberado, permitindo que outro thread use o objeto. Em um momento posterior, o thread adormecido é despertado quando outro thread entra no mesmo bloqueio e chama Pulse() ou PulseAll(). Uma chamada para Pulse() retoma o primeiro thread na fila de threads aguardando pelo bloqueio. Uma chamada para PulseAll sinaliza a liberação do bloqueio para todos os threads em espera.

Você pode ter notado um pequeno problema com o código acima. Se o thread A mantém a lock no objeto key, por que a thread B não é bloqueada quando tenta obter a lock? Obviamente, isso é tratado adequadamente. A chamada Wait no thread A libera o lock antes de aguardar. Isso permite que o thread B adquira o lock e a chamada Pulse. O thread A é retomado, mas precisa aguardar até o thread B liberar o lock, para que ele possa recuperá-lo e concluir a chamada Wait. Observe que Pulse nunca bloqueia.



A ready queue é a coleção de threads que aguardam um lock específico. Os métodos Monitor.Wait introduzem outra fila: waiting queue. Isso é necessário, pois esperar por um Pulse é diferente de esperar para adquirir um lock. Como a fila pronta, a fila de espera é FIFO.



Essas filas podem levar a um comportamento inesperado. Quando Pulse ocorre, o cabeçalho da fila de espera é liberado e adicionado à fila de espera. No entanto, se houver outros threads na fila pronta, eles adquirirão o bloqueio antes do thread lançado. Isso é um problema, porque a linha que adquire o bloqueio pode alterar o estado em que a thread pulsada depende. A solução é usar uma condição while dentro da lockinstrução:

readonly object key = new object();

bool block = true;

// thread A

lock (key)

{

while (block)

Monitor.Wait(key);

block = true;

}

// thread B

lock (key)

{

block = false;

Monitor.Pulse(key);

}

O código acima é realmente uma implementação de um AutoResetEvent. Se você omitir a instrução block = true no thread A, seria a ManualResetEvent. Se você usar um int vez de bool para a condição, seria um Semaphore. Isso mostra o quão versátil é esse padrão, pois mostra o motivo da regra na qual os bloqueios devem ser usados: eles protegem a variável de condição do acesso simultâneo. Bloqueios também são barreiras (barriers) de memória, portanto, você não precisa declarar as variáveis ​​de condição como volatile.

Você tem controle total sobre quais condições usar no loop while, ele resolve os problemas:

1. Se o thread B for executado antes do thread A Wait, o valor Pulse será perdido. No entanto, quando o thread A eventualmente é executado e adquire o bloqueio, a primeira coisa que faz é verificar a condição. O thread B já foi definido block como false, portanto, o thread A continua sem nunca chamar Wait.
2. Resolve o problema das filas. Se o thread A for pulsado, ele sai da fila de espera e é adicionado à fila de espera. Se, no entanto, um thread diferente adquire o bloqueio e esse thread blockvolta true, agora não importa. Isso ocorre porque o thread A contorna o whileloop, encontra a condição para bloquear e é trueexecutado Waitnovamente.

Portanto, a regra geral é: use primitivas de nível superior, se elas se ajustarem. Se você precisar de um controle mais preciso, use os métodos Monitor com esse padrão para garantir a correção.

Geralmente, não há problema em usar PulseAll em vez de Pulse. Isso faz com que todos os threads em espera reavaliem suas condições e continuem ou retornem à espera. Desde que a avaliação da condição não seja muito cara, isso normalmente é aceitável. PulseAll também é útil onde você tem vários threads aguardando o mesmo objeto de sincronização, mas com condições diferentes.

Existem sobrecargas Wait que levam um parâmetro de tempo limite. Semelhante ao acima. Pode ajudar a evitar Waits presos ou perdidos Pulsereavaliando as condições regularmente.

Abaixo está um exemplo com o código fonte completo que demonstra a versatilidade desse padrão. Ele implementa uma fila de bloqueio que pode ser parada. Uma fila de bloqueio é uma fila de tamanho fixo. Se a fila estiver cheia, as tentativas de adicionar um item serão bloqueadas. Se a fila estiver vazia, as tentativas de remover um item serão bloqueadas. Quando Quit() é chamado, a fila é parada. Isso significa que você não pode adicionar mais itens, mas pode remover itens existentes até que a fila esteja vazia. Nesse ponto, a fila está concluída.

Este é um conjunto bastante complexo de condições. Você pode implementar isso usando uma combinação de construções de nível superior, mas seria mais difícil. O padrão torna essa implementação relativamente trivial.

class BlockingQueue<T>

{

readonly int \_Size = 0;

readonly Queue<T> \_Queue = new Queue<T>();

readonly object \_Key = new object();

bool \_Quit = false;

public BlockingQueue(int size)

{

\_Size = size;

}

public void Quit()

{

lock (\_Key)

{

\_Quit = true;

Monitor.PulseAll(\_Key);

}

}

public bool Enqueue(T t)

{

lock (\_Key)

{

while (!\_Quit && \_Queue.Count >= \_Size) Monitor.Wait(\_Key);

if (\_Quit) return false;

\_Queue.Enqueue(t);

Monitor.PulseAll(\_Key);

}

return true;

}

public bool Dequeue(out T t)

{

t = default(T);

lock (\_Key)

{

while (!\_Quit && \_Queue.Count == 0) Monitor.Wait(\_Key);

if (\_Queue.Count == 0) return false;

t = \_Queue.Dequeue();

Monitor.PulseAll(\_Key);

}

return true;

}

}

internal static void Test()

{

var q = new BlockingQueue<int>(4);

// Producer

new Thread(() =>

{

for (int x = 0; ; x++)

{

if (!q.Enqueue(x)) break;

Trace.WriteLine(x.ToString("0000") + " >");

}

Trace.WriteLine("Producer quitting");

}).Start();

// Consumers

for (int i = 0; i < 2; i++)

{

new Thread(() =>

{

for (; ; )

{

Thread.Sleep(100);

int x = 0;

if (!q.Dequeue(out x)) break;

Trace.WriteLine(" < " + x.ToString("0000"));

}

Trace.WriteLine("Consumer quitting");

}).Start();

}

Thread.Sleep(1000);

Trace.WriteLine("Quitting");

q.Quit();

}

**Classe CountdownEvent**

O .NET 4 introduziu uma nova classe chamada CoundownEvent, definida no namespace System.Threading. O cenário de uso é direto: você precisa aguardar um número predefinido de threads para concluir seu trabalho. Antes do .NET 4, isso era implementado usando vários objetos EventWaitHandle e chamando o método WaitHandle.WaitAll. Como esse é um cenário comum, a Microsoft decidiu implementar essa funcionalidade no .NET. As tabelas abaixo listam os métodos e propriedades mais comuns de CoundownEvent.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| CountdownEvent | Construtor que aceita como parâmetro um valor inteiro chamado count, representando o número de sinais que precisa receber antes de ser sinalizado. |
| AddCount | Duas sobrecargas. Aumenta a contagem atual do CountdownEvent em um, ou por um valor especificado. Se o objeto CountdownEvent já estiver definido, esse método poderá gerar uma InvalidOperationException. |
| Dispose | Este é o método da interface IDisposable. Você deve chamar esse método para garantir que os recursos do SO sejam liberados quando esse objeto não for mais necessário. |
| Reset | Duas sobrecargas. Redefine o CurrentCount para o valor de InitialCount ou para um valor especificado. |
| Signal | Duas sobrecargas. Registra um sinal com o CountdownEvent, diminuindo o valor de CurrentCount por um ou por um valor especificado. |
| TryAddCount | Duas sobrecargas. Tenta incrementar CurrentCount por um ou por um valor especificado. Este método não gera uma exceção, como AddCount. Retorna verdadeiro ou falso para indicar o sucesso ou falha da operação. |
| Wait | Seis sobrecargas. Bloqueia o thread atual até que o CountdownEvent seja definido. As sobrecargas são usadas para chamar o método com um token de cancelamento e/ou com um tempo limite. |

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| CurrentCount | Propriedade somente leitura que retorna o número de sinais restantes necessários para definir o evento. |
| InitialCount | Propriedade somente leitura que retorna o número de sinais inicialmente necessários para definir o evento. |
| IsSet | Propriedade somente leitura que retorna true se o evento estiver definido. |
| WaitHandle | Propriedade somente leitura que retorna um WaitHandle usado para aguardar a configuração do evento. |

A classe System.Threading.CountdownEvent representa um evento definido quando a contagem é zero. Embora CountdownEvent.CurrentCount seja maior que zero, um thread que chama CountdownEvent.Wait é bloqueado. Chame CountdownEvent.Signal para diminuir a contagem de um evento.

Em contraste com ManualResetEvent ou ManualResetEventSlim, que você pode usar para desbloquear vários threads com um sinal de um thread, você pode usar CountdownEvent para desbloquear um ou mais threads com sinais de vários threads. CountdownEvent permite esperar mais de um thread. A classe é nova no Framework 4.0 e possui uma implementação eficiente totalmente gerenciada. Se você estiver executando uma versão anterior a este Framework, deverá implementar um CountdownEvent usando Wait e Pulse.

Para usar CountdownEvent, instancie a classe com o número de threads ou "contagens" nas quais você deseja esperar:

var countdown = novo CountdownEvent (3); // Inicialize com "count" de 3.

O sinal de chamada diminui a "contagem"; chamada Espera bloqueia até que a contagem desça para zero. Por exemplo:

static CountdownEvent \_countdown = new CountdownEvent(3);

static void Main(string[] args)

{

new Thread(SaySomething).Start("I am thread 1");

new Thread(SaySomething).Start("I am thread 2");

new Thread(SaySomething).Start("I am thread 3");

\_countdown.Wait(); // Blocks until Signal has been called 3 times

Console.WriteLine("All threads have finished speaking!");

Console.ReadKey();

}

static void SaySomething(object thing)

{

Thread.Sleep(1000);

Console.WriteLine(thing);

\_countdown.Signal();

}

Às vezes, os problemas para os quais o CountdownEvent é eficaz podem ser resolvidos mais facilmente usando as construções de paralelismo estruturado (PLINQ e a classe Parallel).

Você pode recriar a contagem de um CountdownEvent ligando para AddCount. No entanto, se já atingiu zero, isso gera uma exceção: você não pode "cancelar o sinal" de um CountdownEvent chamando AddCount. Para evitar a possibilidade de uma exceção ser lançada, você pode chamar TryAddCount, que retornará false se a contagem regressiva for zero. Para cancelar a sinalização de um evento de contagem regressiva, chame Reset: isso cancela a sinalização da construção e redefine sua contagem para o valor original.

Como ManualResetEventSlim, CountdownEvent expõe uma propriedade WaitHandle para cenários em que alguma outra classe ou método espera um objeto com base em WaitHandle.

**Classe Barrier**

Em um cenário multithread, há situações em que você gera vários threads e deseja garantir que eles cheguem todos em um determinado ponto antes que você possa continuar a execução do seu código. Um exemplo comum para esse cenário é o seguinte: Um grupo de amigos decide viajar de carro do ponto A ao ponto C, via ponto B. Eles querem começar a viajar juntos do ponto A e parar no ponto B; eles planejam começar juntos novamente para viajar e se encontrar no ponto final C. Alguns deles podem até decidir que não querem mais ir e voltar para casa.

A classe System.Threading.Barrier representa uma barreira de execução do thread. Um thread que chama o método Barrier.SignalAndWait sinaliza que atingiu a barreira e aguarda até que outros threads participantes atinjam a barreira. Quando todos os threads participantes atingem a barreira, eles prosseguem e a barreira é redefinida e pode ser usada novamente.

Antes de analisar uma possível implementação, você precisa examinar o que o .NET pode oferecer para resolver esse tipo de problema. Uma maneira seria usar o evento Countdown, mas na verdade não está modelando o que você precisa! O .NET 4 introduziu uma nova classe chamada System.Threading.Barrier que lida com exatamente essas situações. As tabelas abixo listam alguns dos métodos e propriedades da classe.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| Barrier Constructor | Inicializa uma nova instância da classe Barrier. Este método possui duas sobrecargas, ambas tomando como parâmetro o número de participantes.  A segunda sobrecarga usa como parâmetro extra uma Ação que será executada após todos os participantes terem chegado à barreira em uma fase. |
| AddParticipant | Envie uma notificação para a Barreira de que haverá mais um participante. |
| AddParticipants | Envie uma notificação para a Barreira de que haverá vários outros participantes. |
| Dispose | Este é o método da interface IDisposable. Você deve chamar esse método para garantir que os recursos do SO sejam liberados quando esse objeto não for mais necessário. |
| RemoveParticipant | Envia uma notificação para a Barreira de que haverá menos um participante. |
| RemoveParticipants | Envia uma notificação para a Barreira de que haverá menos participantes. |
| SignalAndWait | Seis sobrecargas. Sinais de que um participante alcançou a barreira e espera que todos os outros participantes também atinjam a barreira. As sobrecargas são usadas para chamar o método com um token de cancelamento e / ou com um tempo limite. |

Propriedades de System.Threading.Barrier

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| CurrentPhaseNumber | Propriedade somente leitura que retorna o número da fase atual da barreira. |
| ParticipantCount | Propriedade somente leitura que retorna o número total de participantes na barreira. |
| ParticipantsRemaining | Propriedade somente leitura que retorna o número de participantes na barreira que ainda não chegou. |

Considere o seguinte snippet de código que usa a classe Barrier:

var participants = 5;

Barrier barrier = new Barrier(participants + 1,

// We add one for the main thread.

b =>

{ // This method is only called when all the paricipants arrived.

Console.WriteLine("{0} paricipants are at rendez-vous point {1}.",

b.ParticipantCount - 1, // We substract the main thread.

b.CurrentPhaseNumber);

});

for (int i = 0; i < participants; i++)

{

var localCopy = i;

Task.Run(() =>

{

Console.WriteLine("Task {0} left point A!", localCopy);

Thread.Sleep(1000 \* localCopy + 1); // Do some "work"

if (localCopy % 2 == 0)

{

Console.WriteLine("Task {0} arrived at point B!", localCopy);

barrier.SignalAndWait();

}

else

{

Console.WriteLine("Task {0} changed mind and went back!", localCopy);

barrier.RemoveParticipant();

return;

}

Thread.Sleep(1000 \* (participants - localCopy)); // Do some "more work"

Console.WriteLine("Task {0} arrived at point C!", localCopy);

barrier.SignalAndWait();

});

}

Console.WriteLine("Main thread is waiting for {0} tasks!",

barrier.ParticipantCount - 1);

barrier.SignalAndWait(); // Waiting at the first phase

barrier.SignalAndWait(); // Waiting at the second phase

Console.WriteLine("Main thread is done!");

Nesta amostra, você cria uma barreira para acompanhar os participantes que chegaram aos pontos de reunião. Você precisa inicializar a barreira com o número de participantes mais um. O participante extra é usado pelo thread principal.

Então você cria uma tarefa por participante. A instrução var localCopy = i captura o valor do iterador, para evitar problemas que possam aparecer. Apenas para tornar o cenário mais interessante, todas as outras tarefas “mudarão de idéia” e voltarão, mas não antes de informar as outras. O thread principal está chamando de barreira.SignalAndWait duas vezes, uma vez para cada fase.

**CONSTRUÇÕES DE SINCRONIZAÇÃO SEM BLOQUEIO**

Anteriormente, dissemos que a necessidade de sincronização surge mesmo no simples caso de atribuir ou incrementar um campo. Embora o bloqueio sempre possa satisfazer essa necessidade, um bloqueio contido significa que um thread deve bloquear, sofrendo a sobrecarga de uma alternância de contexto e a latência de ser programado, o que pode ser indesejável em cenários altamente simultâneos e críticos para o desempenho. As construções de sincronização sem bloqueio do .NET Framework podem executar operações simples sem nunca bloquear, pausar ou aguardar.

Escrever código multithread nonblocking ou lock-free corretamente é complicado. As Memory barriers, em particular, são fáceis de errar (a palavra-chave volátil é ainda mais fácil de errar). Pense com cuidado se você realmente precisa dos benefícios de desempenho antes de descartar bloqueios comuns. Lembre-se de que adquirir e liberar um bloqueio não-necessário leva apenas 20ns em dias atuais.

As abordagens nonblocking também funcionam em vários processos. Um exemplo de onde isso pode ser útil está na leitura e gravação de memória compartilhada do processo.

class Foo

{

public int \_answer;

bool \_complete;

public void A()

{

\_answer = 123;

\_complete = true;

}

public void B()

{

if (\_complete) Console.WriteLine(\_answer);

}

}

Se os métodos A e B forem executados simultaneamente em diferentes threads, será possível que B escreva “0”? A resposta é sim - pelos seguintes motivos:

* O compilador, CLR ou CPU pode reordenar as instruções do seu programa para melhorar a eficiência.
* O compilador, CLR ou CPU pode introduzir otimizações de cache de forma que atribuições a variáveis ​​não sejam visíveis para outros threads imediatamente.

Trabalhar com campos graváveis compartilhados sem bloqueios ou cercas está pedindo por problemas. Há muitas informações enganosas sobre esse tópico - incluindo a documentação do MSDN, que afirma que o MemoryBarrier é necessário apenas em sistemas multiprocessadores com pouca memória, como um sistema que utiliza vários processadores Itanium. Podemos demonstrar que as barreiras de memória são importantes nos processadores Intel Core-2 e Pentium comuns com o seguinte programa curto. Você precisará executá-lo com as otimizações ativadas e sem um depurador (no Visual Studio, selecione Release Mode no gerenciador de configuração da solução e inicie sem depuração):

bool complete = false;

var t = new Thread(() =>

{

bool toggle = false;

while (!complete)

{

Console.WriteLine(!complete);

toggle = !toggle;

}

});

t.Start();

Thread.Sleep(1000);

complete = true;

t.Join(); // Blocks indefinitely

Este programa nunca termina porque a variável complete é armazenada em cache em um registro da CPU. Inserir uma chamada para Thread.MemoryBarrier dentro do loop while (ou travar a leitura concluída) corrige o erro.

Uma boa abordagem é começar colocando barreiras de memória antes e depois de cada instrução que lê ou grava um campo compartilhado e depois retira os que você não precisa. Se você não tiver certeza, deixe-o dentro. Ou melhor: volte a usar bloqueios!

**Thread.MemoryBarrier**

O C# e o tempo de execução são muito cuidadosos para garantir que essas otimizações não quebrem o código comum de thread único - ou o código multithread que faz uso adequado dos bloqueios. Fora desses cenários, você deve derrotar explicitamente essas otimizações criando barreiras de memória (também chamadas cercas de memória) para limitar os efeitos do reordenamento de instruções e do cache de leitura / gravação.

O tipo mais simples de barreira de memória é uma barreira de memória completa (full fence) que impede qualquer tipo de instrução reordenar ou armazenar em cache em torno dessa barreira. Chamar Thread.MemoryBarrier gera uma cerca completa; podemos corrigir nosso exemplo aplicando quatro cercas completas da seguinte maneira:

public void MemoryBarrier\_A()

{

\_answer = 123;

Thread.MemoryBarrier(); // Barrier 1

\_complete = true;

Thread.MemoryBarrier(); // Barrier 2

}

public void MemoryBarrier\_B()

{

Thread.MemoryBarrier(); // Barrier 3

if (\_complete)

{

Thread.MemoryBarrier(); // Barrier 4

Console.WriteLine(\_answer);

}

}

As barreiras 1 e 4 impedem que este exemplo escreva "0". As barreiras 2 e 3 fornecem uma garantia de atualização: elas garantem que se B for executado após A, a leitura de \_complete será avaliada como verdadeira.

Uma cerca completa (full fence) leva cerca de dez nanossegundos em um desktop. O seguinte gera implicitamente cercas completas:

* Instrução de bloqueio do C # (Monitor.Enter / Monitor.Exit)
* Todos os métodos da classe Interlocked
* Retornos de chamada assíncronos que usam o conjunto de threads - incluem delegados assíncronos, retornos de chamada APM e continuações de tarefas
* Qualquer coisa que dependa de sinalização, como iniciar ou aguardar uma tarefa. Em virtude desse último ponto, o seguinte é seguro para threads:

int x = 0;

Task t = Task.Factory.StartNew (() => x++);

t.Wait();

Console.WriteLine (x); // 1

Você não precisa necessariamente de uma cerca completa com cada leitura ou gravação individual. Se tivéssemos três campos de resposta, nosso exemplo ainda precisaria de apenas quatro cercas:

class Foo\_Partial

{

int \_answer1, \_answer2, \_answer3;

bool \_complete;

void A()

{

\_answer1 = 1; \_answer2 = 2; \_answer3 = 3;

Thread.MemoryBarrier();

\_complete = true;

Thread.MemoryBarrier();

}

void B()

{

Thread.MemoryBarrier();

if (\_complete)

{

Thread.MemoryBarrier();

Console.WriteLine(\_answer1 + \_answer2 + \_answer3);

}

}

}

**Classe Volatile**

Outra maneira (mais avançada) de resolver esse problema apresentado anteriormente relacionado ao campo \_complete é utilizando outra classe no .NET Framework: System.Threading.Volatile. Essa classe possui um método especial de gravação e leitura, e esses métodos desabilitam as otimizações do compilador para que você possa forçar a ordem correta no seu código. O uso desses métodos na ordem correta pode ser bastante complexo, portanto, o .NET oferece a palavra-chave volátil que você pode aplicar a um campo. Você então alteraria a declaração do seu campo para isto:

volatile bool \_complete;

A palavra-chave volátil instrui o compilador a gerar uma cerca de aquisição em todas as leituras desse campo e uma cerca de liberação em todas as gravações nesse campo. Uma cerca de aquisição impede que outras leituras/gravações sejam movidas antes da cerca; uma barreira de liberação impede que outras leituras/gravações sejam movidas após a barreira. Essas “meias cercas” são mais rápidas que as cercas completas, porque dão ao tempo de execução e ao hardware mais espaço para otimização. O efeito da aplicação de voláteis a campos pode ser resumido da seguinte forma:

|  |  |  |
| --- | --- | --- |
| **First instruction** | **Second instruction** | **Can they be swapped?** |
| Read | Read | No |
| Read | Write | No |
| Write | Write | No (The CLR ensures that write-write operations are never swapped, even without the volatile keyword) |
| Write | Read | Yes! |

Observe que a aplicação de volátil não impede que uma gravação seguida de uma leitura seja trocada, e isso pode criar quebra-cabeças. Joe Duffy ilustra bem o problema com o seguinte exemplo: se Test1 e Test2 forem executados simultaneamente em threads diferentes, é possível que a e b terminem com o valor 0 (apesar do uso de voláteis em x e y):

class IfYouThinkYouUnderstandVolatile

{

volatile int x, y;

void Test1() // Executed on one thread

{

x = 1; // Volatile write (release-fence)

int a = y; // Volatile read (acquire-fence)

}

void Test2() // Executed on another thread

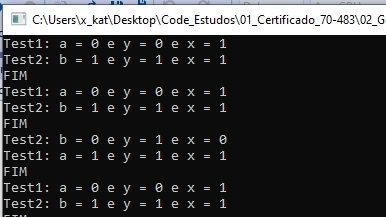
{

y = 1; // Volatile write (release-fence)

int b = x; // Volatile read (acquire-fence)

}

}



Isso representa um forte argumento para evitar a volatilidade: mesmo se você entender a sutileza neste exemplo, outros desenvolvedores trabalhando no seu código também a entenderão? Uma cerca completa entre cada uma das duas atribuições no Teste1 e Teste2 (ou uma trava) resolve o problema.

É bom estar ciente da existência da palavra-chave volátil, mas é algo que você deve usar apenas se realmente precisar. Como desativa certas otimizações do compilador, isso prejudica o desempenho. Também não é algo suportado por todas as linguagens .NET (o Visual Basic não suporta), por isso dificulta a interoperabilidade da linguagem.

A palavra-chave volátil não é suportada com argumentos de passagem por referência ou variáveis locais capturadas: nesses casos, você deve usar os métodos VolatileRead e VolatileWrite.

**VolatileRead e VolatileWrite**

Os métodos estáticos VolatileRead e VolatileWrite na classe Thread leem/gravam uma variável enquanto aplicam (tecnicamente, um superconjunto) as garantias feitas pela palavra-chave volátil. Suas implementações são relativamente ineficientes, pois geram cercas completas.

**Interlocked**

Tornar as operações atômicas é o trabalho da classe Interlocked que pode ser encontrada no namespace System.Threading. A classe Interlocked é usada para sincronizar o acesso de objetos de memória compartilhada entre vários threads. A classe intertravada fornece a seguinte operação útil na memória compartilhada:

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| Add | Adiciona dois inteiros de 32 ou 64 bits e substitui o primeiro inteiro pela soma, como uma operação atômica. |
| CompareExchange | Compara o primeiro e o terceiro parâmetros de igualdade e, se forem iguais, substitui o valor do primeiro parâmetro pelo segundo parâmetro. |
| Decrement | Decrementa uma variável especi ﬁ cada e armazena o resultado, como uma operação atômica. |
| Exchange | Define um objeto para um valor especificado e retorna uma referência ao objeto original, como uma operação atômica. |
| Increment | Incrementa uma variável especi ﬁ cada e armazena o resultado, como uma operação atômica. |
| Read | Carrega um valor de 64 bits como uma operação atômica e o retorna ao chamador. Isso é necessário apenas em plataformas de 32 bits |

No exemplo anterior, o problema essencial era que as operações de adição(n = n + 1 ou n++) e subtração (num = num – 1 ou n--) não eram atômicas tanto para leitura quanto para gravação. Ao utilizar o Interlocked ficaria o seguinte:

for (int i = 0; i < length; i++)

{

Interlocked.Decrement(ref num); //Ou Interlocked.Increment(ref num);

}

O Interlocked garante que as operações de incremento e decremento sejam executadas atomicamente. Nenhum outro thread verá resultados intermediários. Obviamente, adicionar e subtrair é uma operação simples. Se você tiver operações mais complexas, ainda precisará usar um lock.

O Interlocked também oferece suporte à alternância de valores usando o método Exchange. Você usa esse método da seguinte maneira:

if (Interlocked.Exchange(ref n, 1) == 0) { }

Esse código recupera o valor atual e o define imediatamente para o novo valor na mesma operação. Retorna o valor anterior antes de alterá-lo. A seguir um exemplo de uso de Interlocked.Exchange:

//0 for false, 1 for true.

private static int usingResource = 0;

private const int numThreadIterations = 5;

private const int numThreads = 10;

static void Main(string[] args)

{

Thread myThread;

Random rnd = new Random();

for (int i = 0; i < numThreads; i++)

{

myThread = new Thread(new ThreadStart(MyThreadProc));

myThread.Name = String.Format("Thread{0}", i + 1);

//Wait a random amount of time before starting next thread.

Thread.Sleep(rnd.Next(0, 1000));

myThread.Start();

}

Console.ReadKey();

}

private static void MyThreadProc()

{

for (int i = 0; i < numThreadIterations; i++)

{

UseResource();

//Wait 1 second before next attempt.

Thread.Sleep(1000);

}

}

//A simple method that denies reentrancy.

static bool UseResource()

{

//0 indicates that the method is not in use.

if (0 == Interlocked.Exchange(ref usingResource, 1))

{

Console.WriteLine("{0} acquired the lock", Thread.CurrentThread.Name);

//Code to access a resource that is not thread safe would go here.

//Simulate some work

Thread.Sleep(500);

Console.WriteLine("{0} exiting lock", Thread.CurrentThread.Name);

//Release the lock

Interlocked.Exchange(ref usingResource, 0);

return true;

}

else

{

Console.WriteLine("{0} was denied the lock", Thread.CurrentThread.Name);

return false;

}

}

As operações matemáticas do Interlocked estão restritas a Incremento, Decremento e Adição. Se você deseja multiplicar - ou executar qualquer outro cálculo - você pode fazê-lo no estilo livre de bloqueio, usando o método CompareExchange (normalmente em conjunto com a espera em rotação). Este método primeiro verifica se o valor esperado está lá; se for, substitui-o por outro valor. O exmplo abaixo mostra o que pode dar errado ao comparar e trocar um valor em uma operação não atômica.

int value = 1;

Task t1 = Task.Run(() =>

{

if (value == 1)

{

// Removing the following line will change the output

Thread.Sleep(1000);

value = 2;

}

});

Task t2 = Task.Run(() =>

{

value = 3;

});

Task.WaitAll(t1, t2);

Console.WriteLine(value); // Displays 2

A tarefa t1 começa a executar e vê que o valor é igual a 1. Ao mesmo tempo, t2 altera o valor para 3 e, em seguida, t1 o altera de volta para 2. Para evitar isso, você pode usar bloqueios:

object \_lock = new object();

Task t1 = Task.Run(() =>

{

lock (\_lock)

{

if (value == 1)

{

// Removing the following line will change the output

Thread.Sleep(1000);

value = 2;

}

}

});

Task t2 = Task.Run(() =>

{

lock (\_lock)

{

value = 3;

}

});

Ou com a seguinte instrução Interlocked:

Interlocked.CompareExchange (ref value, newValue, compareTo);

Que basicamente faz isso:

if(value==compareTo)

{

value = newValue;

}

Task t1 = Task.Run(() =>

{

Interlocked.CompareExchange(ref value, 2, 1);

});

Isso garante que comparar o valor e trocá-lo por um novo seja uma operação atômica. Dessa forma, nenhum outro thread pode alterar o valor entre compará-lo e trocá-lo. Interlocked.CompareExchange atualiza um campo com um valor especificado se o valor atual do campo corresponder ao terceiro argumento. Em seguida, ele retorna o valor antigo do campo, para que você possa testar se ele foi bem-sucedido comparando-o com o instantâneo original. Se os valores diferirem, isso significa que outro thread o antecipou; nesse caso, você gira e tenta novamente.

Todos os métodos da Interlocked geram uma barreira completa. Portanto, os campos que você acessa via Interlocked não precisam de cercas adicionais, a menos que sejam acessados ​​em outros lugares do programa sem o Interlocked ou um bloqueio. Os métodos de intertravamento têm uma sobrecarga típica de 10 ns - metade da de um bloqueio não controlado. Além disso, eles nunca podem sofrer o custo adicional da alternância de contexto devido ao bloqueio. O outro lado é que usar o Interlocked dentro de um loop com muitas iterações pode ser menos eficiente do que obter um único bloqueio ao redor do loop (embora o Interlocked permita maior simultaneidade).

**System.Threading.CancellationTokenSource**

A classe Task da suporte a cancelamento cooperativo e é totalmente integrado com a classe System.Threading.CancellationTokenSource e com a classe System.Threading.CancellationToken, que são novos no Framework 4. NET. Muitos dos construtores da classe System.Threading.Tasks.Task tem um CancellationToken como parâmetro de entrada. Muitas das sobrecargas StartNew e Run também possuem um CancellationToken.

**Cancelando Tasks**

A partir do .NET Framework 4, o .NET Framework usa um modelo unificado para cancelamento cooperativo de operações assíncronas ou síncronas de longa execução. Este modelo é baseado em um objeto leve chamado token de cancelamento. Antes do .NET 4, as formas de cancelar uma operação em andamento eram inseguras. Eles incluíam abortar e interromper threads ou até abandonar operações nas quais você não estava mais interessado. Embora isso funcionasse na maioria das vezes, os cancelamentos foram a fonte de muitos erros.

Os cancelamentos fornecidos no .NET são cancelamentos cooperativos, o que significa que você pode enviar uma solicitação de cancelamento para outro thread, ou tarefa, mas é sua escolha atender à solicitação. Os recursos de cancelamento são implementados usando uma classe, CancellationTokenSource e uma estrutura, CancellationToken. O objeto solicitante invoca uma ou mais operações canceláveis, por exemplo criando novos threads, itens de trabalho do ThreadPool ou tarefas, passa o token para cada operação. As operações individuais podem, por sua vez, passar cópias do token para outras operações. Posteriormente, o objeto solicitante que criou o token pode usá-lo para solicitar que as operações parem o que estão fazendo. Somente o objeto solicitante pode emitir a solicitação de cancelamento, e cada ouvinte é responsável por perceber a solicitação e respondê-la de forma apropriada e oportuna.

A estrutura de cancelamento é implementada como um conjunto de tipos relacionados, que estão listados na tabela a seguir.

|  |  |
| --- | --- |
| **Nome de tipo** | **Descrição** |
| [CancellationTokenSource](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.cancellationtokensource) | O objeto que cria um token de cancelamento, e também emite o pedido de cancelamento para todas as cópias desse token. |
| [CancellationToken](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.cancellationtoken) | O tipo de valor leve passado a um ou mais ouvintes, normalmente como um parâmetro de método. Os ouvintes monitoram o valor da propriedade IsCancellationRequested do token por sondagem, retorno de chamada ou identificador de espera. |
| [OperationCanceledException](https://docs.microsoft.com/pt-br/dotnet/api/system.operationcanceledexception) | As sobrecargas do construtor desta exceção aceitam [CancellationToken](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.cancellationtoken) como um parâmetro. Os ouvintes podem, opcionalmente, lançar essa exceção para verificar a origem do cancelamento e notificar aos outros que ela respondeu a uma solicitação de cancelamento. |

O cancelamento ocorre ao solicitar um código que chama o método CancellationTokenSource.Cancel e o delegado do usuário finaliza a operação. No entanto, uma operação pode ser finalizada:

1. Simplesmente retornando do delegado;
2. Chamando o método CancellationTokenSource.Cancel.

A seguir, são etapas gerais para implementar o modelo de cancelamento:

1. **Instancie um CancellationTokenSource**, se um thread deseja ter a capacidade de cancelar operações subseqüentes precisa de um objeto [CancellationTokenSource](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.cancellationtokensource), que gerencia e envia uma notificação de cancelamento para os tokens de cancelamento individuais.
2. **Obtenha um CancellationToken** da propriedade CancellationTokenSource.Token.
3. **Passe o CancellationToken** para cada tarefa ou thread que escuta o cancelamento.
4. **Forneça um mecanismo para cada tarefa ou thread** para responder ao cancelamento. Operação assíncrona cancelável significa uma operação que oferece suporte para cancelamentos ou um novo thread que será criada pelo thread atual. Isso normalmente é expresso na forma de um ou vários métodos sobrecarregados que aceitam um CancellationToken .
5. **Chame o método CancellationTokenSource.Cancel** para fornecer uma notificação de cancelamento que cancela as operações canceláveis ​​em andamento. Todas as operações em andamento podem usar o CancelationToken enviado como parâmetro para verificar se um cancelamento está pendente e responder de acordo ou ignorar a solicitação.

A classe [CancellationTokenSource](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.cancellationtokensource) implementa a interface [IDisposable](https://docs.microsoft.com/pt-br/dotnet/api/system.idisposable). Certifique-se de chamar o método [CancellationTokenSource.Dispose](https://docs.microsoft.com/pt-br/dotnet/api/system.threading.cancellationtokensource.dispose) quando terminar de usar a fonte de token de cancelamento para liberar todos os recursos não gerenciados detidos.

//1 - Instantiate a cancellation token source

using (CancellationTokenSource cts = new CancellationTokenSource())

{

//2 - Get token from CancellationTokenSource.Token property

CancellationToken token = cts.Token;

//3 - Pass token to Task

Task task = Task.Run(() =>

{

//4 - Mecanismo para cada tarefa ou thread para responder ao cancel

while (!token.IsCancellationRequested)

{

Console.Write("\*");

Thread.Sleep(1000);

}

}, token);

Console.WriteLine("Press enter to stop the task");

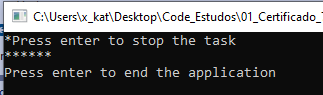
Console.ReadLine();

//5 - notify for cancellation

cts.Cancel();

}

Console.WriteLine("Press enter to end the application");



O CancellationToken é usado na tarefa assíncrona. O CancellationTokenSource é usado para sinalizar que a tarefa deve se cancelar. Nesse caso, a operação terminará quando o cancelamento for solicitado. Usuários externos da tarefa não verão nada diferente, pois a tarefa terá apenas um estado **RanToCompletion**. Se você deseja sinalizar para usuários externos que sua tarefa foi cancelada, você pode fazer isso lançando uma OperationCanceledException. O exemplo abaixo mostra como fazer isso.

using (CancellationTokenSource cts = new CancellationTokenSource())

{

CancellationToken token = cts.Token;

Task task = Task.Run(() =>

{

while (!token.IsCancellationRequested)

{

Console.Write("\*");

Thread.Sleep(1000);

token.ThrowIfCancellationRequested();

}

}, token);

try

{

Console.WriteLine("Press enter to stop the task");

Console.ReadLine();

cts.Cancel();

task.Wait();

}

catch (AggregateException e)

{

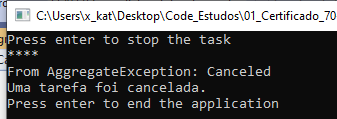
Console.WriteLine("From AggregateException: " + task.Status);

Console.WriteLine(e.InnerExceptions[0].Message);

}

};

Console.WriteLine("Press enter to end the application");



Em vez de capturar a exceção, você também pode adicionar uma tarefa de continuação que é executada apenas quando a tarefa é cancelada. Nesta tarefa, você tem acesso à exceção lançada e pode resolvê-la, se apropriado. O exemplo a seguir mostra como seria uma tarefa de continuação.

//3 - Pass token to Task

Task task = Task.Run(() =>

{

while (!token.IsCancellationRequested)

{

Console.Write("\*");

Thread.Sleep(1000);

}

}, token).ContinueWith((t) =>

{

Console.WriteLine("From Continuation: " + t.Status);

Console.WriteLine("You have canceled the task");

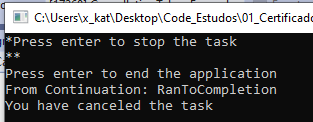
}, TaskContinuationOptions.OnlyOnRanToCompletion);

Console.WriteLine("Press enter to stop the task");

Console.ReadLine();

//5 - notify for cancellation

cts.Cancel();



Observe que, embora .Cancel() tenha sido chamado, o task.Status da continuação é RanToCompletion. Observe também que nenhuma AggregationException é lançada. Isso mostra que apenas chamar .Cancel() da fonte do token não define o status da tarefa como Cancelado. Para o tratamento com o ContinueWith seja utilizado com um tratamento TaskContinuationOptions.OnlyOnCanceled

Task task = Task.Run(() =>

{

//4 - Mecanismo para cada tarefa ou thread para responder ao cancel

while (!token.IsCancellationRequested)

{

Console.Write("\*");

Thread.Sleep(1000);

token.ThrowIfCancellationRequested();

}

}, token);

task.ContinueWith((t) =>

{

Console.WriteLine("From Continuation: " + t.Status);

Console.WriteLine("You have canceled the task");

}, TaskContinuationOptions.OnlyOnCanceled);

Console.WriteLine("Press enter to stop the task");

Console.ReadLine();

//5 - notify for cancellation

cts.Cancel();

Se você deseja cancelar uma tarefa após um certo período de tempo, pode usar uma sobrecarga de Task.WaitAny que leva um tempo limite.

//5 - notify for cancellation

cts.CancelAfter(5000);

Thread.Sleep(5000);

Console.WriteLine();

Console.WriteLine("Task timed out after 5s: " + task.Status);

Vimos que Barriers podem ser usadas para coordenar a chegada de vários threads no mesmo ponto. Mas o que acontece se você quiser cancelar o rocedimento? O próximo exemplo de código faz exatamente isso.

var participants = 5;

// We create a CancellationTokenSource to be able to initiate the cancellation

var tokenSource = new CancellationTokenSource();

// We create a barrier object to use it for the rendez-vous points

var barrier = new Barrier(participants,

b =>

{

Console.WriteLine("{0} paricipants are at rendez-vous point {1}.",

b.ParticipantCount,

b.CurrentPhaseNumber);

});

for (int i = 0; i < participants; i++)

{

var localCopy = i;

Task.Run(() =>

{

Console.WriteLine("Task {0} left point A!", localCopy);

Thread.Sleep(1000 \* localCopy + 1); // Do some "work"

if (localCopy % 2 == 0)

{

Console.WriteLine("Task {0} arrived at point B!", localCopy);

barrier.SignalAndWait(tokenSource.Token);

}

else

{

Console.WriteLine("Task {0} changed its mind and went back!",

localCopy);

barrier.RemoveParticipant();

return;

}

Thread.Sleep(1000 \* localCopy + 1);

Console.WriteLine("Task {0} arrived at point C!", localCopy);

barrier.SignalAndWait(tokenSource.Token);

});

}

Console.WriteLine("Main thread is waiting for {0} tasks!",

barrier.ParticipantsRemaining - 1);

Console.WriteLine("Press enter to cancel!");

Console.ReadLine();

if (barrier.CurrentPhaseNumber < 2)

{

tokenSource.Cancel();

Console.WriteLine("We canceled the operation!");

}

else

{

Console.WriteLine("Too late to cancel!");

}

Console.WriteLine("Main thread is done!");

Implementar fluxo de programa

* Iterar em itens de coleção e de matriz; programar decisões usando instruções switch, se/então e operadores; avaliar expressões

Você pode entender como controlar o fluxo do programa usando estruturas de decisão e repetição. Esses são os principais componentes de um aplicativo escrito em C. O C# oferece algumas instruções que podem ser usadas quando você precisa tomar uma decisão em seu aplicativo, incluindo verificar se o usuário digitou a senha correta, garantir que um determinado valor esteja dentro do alcance ou uma das inúmeras outras possibilidades.

Além de tomar decisões, outra tarefa comum é trabalhar com coleções. O C# possui recursos que ajudam a trabalhar com coleções, permitindo que você itere sobre coleções e acesse itens individuais.

Em qualquer linguagem de programação, declarações são as construções de código que fazem com que o aplicativo execute uma ação. Em C# divide as declarações em dois tipos básicos:.

1. **Instruções simples:** são aquelas que terminam com ponto-e-vírgula (;) e são normalmente usadas para ações do programa, como as seguintes:
   1. Declarando variáveis (declarações)
   2. Atribuindo valores a variáveis (instruções de atribuição)
   3. Chamando o método no seu código
   4. Instruções de ramificação que alteram o fluxo do programa
2. **Instruções complexas:** são aquelas que podem ou incluirão uma ou mais instruções simples em um bloco de código cercado por chaves: {}. Instruções complexas típicas são aquelas que são loops e estruturas de decisão abordadas a seguir como foreach (), if (), switch, do () e assim por diante.

O C# oferece várias instruções de controle de fluxo que ajudam a determinar o caminho que seu aplicativo segue. Normalmente, todas as instruções de um programa são executadas a partir de de cima para baixo. O fluxo de controle ajuda nosso programa a executar ou pular um bloco de código, ajuda a repetir um código até que uma condição seja satisfeita e ajuda nosso controle a saltar para qualquer lugar do código. Mas em um aplicativo real, controlamos o fluxo de execução, apresentando:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Estrutura** | **Instruções** | | | |
| Decisão/Condição | If else | switch | coalescência nula (??) | condicional ternário (? :) |
| Repetição | for | foreach | while | do while |
| Jump | break | goto | continue |  |

Usando essas construções, você pode criar aplicativos flexíveis que permitem executar comportamentos diferentes, dependendo das circunstâncias

**ESTRUTURA DECISÃO/CONDIÇÃO**

Instruções condicionais em C # são aquelas que avaliam uma condição e, em seguida, executam uma ação, não executam nenhuma ação ou escolhem entre as ações disponíveis para execução. Para avaliar condições, o C # fornece o seguinte:

* **Operadores relacionais**
* **Expressions Expressões booleanas**
* **Operadores lógicos**
* **Operator Um operador condicional (operador ternário)**

As condições no seu programa C# permitem comparar valores, normalmente mantidos em variáveis, mas também constantes e literais. Uma variável é um local nomeado na memória que permite armazenar um valor para uso posterior. Isso é chamado de variável porque você pode alterar o conteúdo sempre que quiser. Uma constante é como uma variável, pois é um local de memória nomeado usado para armazenar um valor, mas você não pode alterar o valor à vontade. Ele aceita um valor quando você o declara e mantém esse valor durante toda a vida útil do tempo de execução do seu programa. Literais são valores que, literalmente, são o que são. Exemplos de literais são 1, 25, 'c' e "strings". Você não pode e não atribui outros itens a literais; você pode atribuir literais apenas a variáveis ​​de constantes.

A execução do seu programa pode ser controlada com base nessas comparações. Para usar efetivamente esses conceitos em seus programas, você precisa entender os operadores lógicos de comparação disponíveis (os operadores executam uma operação com valores). A tabela a seguir mostra os operadores relacionais e de igualdade em C#.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Descrição** | **Example** |
| < | Menor que | x < 42; |
| > | Maior que | x > 42; |
| <= | Menos que ou igual a | x <= 42; |
| >= | Melhor que ou igual a | x >= 42; |
| == | Igual a | x == 42; |
| != | Diferente de | x != 42; |

Sempre que vir o operador = em C#, lembre-se de que ele é um operador de atribuição e não um operador de comparação. O C# usa dois sinais = juntos (==) para denotar igualdade. Portanto, 2 = 2 não é o mesmo que 2 == 2. O primeiro não é realmente aceito em C# porque tenta atribuir um literal a um literal, o que não é possível. Um literal em C # é um valor real em oposição a uma variável. No entanto, 2 == 2 é válido em C# e está avaliando se o literal 2 é igual ao literal 2. Nesse caso, é e o resultado é um valor true para a comparação. O operador relacional final é o operador!=, O que significa que não é igual. A expressão 2 != 42 retornaria true porque o valor literal 2 não é igual ao valor literal 42.

**Expressões Booleanas**

Ao trabalhar com instruções de controle de fluxo, você trabalhará automaticamente com expressões booleanas. Uma expressão booleana sempre deve produzir verdadeiro ou falso como resultado final, mas, ao fazer isso, pode ser bastante complexa usando operadores diferentes.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Descrição** | **Example** |
| & | Variante unária retorna o endereço de seu operando. A variante binária é o AND bit a bit de dois operandos. | & expr1 |
| expr1 & expr2 |
| | | O operador OR binário. Verdadeiro se um ou ambos os operandos forem verdadeiros, falso se os dois operandos forem falsos. | expr1 | expr2 |
| ^ | O OR bit a bit exclusivo. Retorna true se, e somente se, um dos operandos for verdadeiro. | expr1 ^ expr2 |
| ! | Operador de negação lógica unário. Retorna false se o operando for verdadeiro ou vice-versa. | !expr |
| ~ | O operador de complemento bit a bit. | ~expr |
| && | Condicional AND que executa uma operação lógica nos operandos booleanos. Capaz de lógica de curto-circuito, em que o segundo operando é avaliado apenas se necessário | expr && expr2 |
| || | Condicional OR que executa um OR lógico nos operandos booleanos. Avalia apenas o segundo operando, se necessário. | expr1 || expr2 |
| true | Usado como um operador bool para indicar a verdade em uma expressão. | bool success = true; |
| false | sado como um operador bool para indicar mentira em uma expressão. | bool success = false; |

Você pode combinar esses operadores usando os operadores OR (||), AND (&&) e OR exclusivo (^). Esses operadores usam um operando esquerdo e um direito, significando a parte esquerda e direita da expressão. O operador OR retorna true quando um dos dois operandos for true. Se ambos forem falsos, retornará falso. Se ambos forem verdadeiros, ele retornará verdadeiro. O código abaixo mostra um exemplo.

bool x2 = true;

bool y2 = false;

bool result = x2 || y2;

Console.WriteLine(result); // Displays True

Se o tempo de execução perceber que a parte esquerda da sua operação OR é verdadeira, não será necessário avaliar a parte direita da sua expressão. Isso é chamado de curto-circuito. O exemplo abaixo mostra um exemplo.

public static void OrShortCircuit()

{

bool x = true;

bool result = x || GetY();

}

private static bool GetY()

{

Console.WriteLine("This method doesn’t get called");

return true;

}

Nesse caso, o método GetY nunca é chamado e a linha não é gravada no console. O operador AND pode ser usado quando as duas partes de uma expressão precisam ser verdadeiras. Se um dos operandos for falso, a expressão inteira será avaliada como falsa. O código a seguir usa o operador AND para verificar se um valor está dentro de um determinado intervalo.

int value = 42;

bool result2 = (0 < value) && (value < 100);

Nesse caso, não é necessário adicionar parênteses extras ao redor do operando esquerdo e direito, mas isso aumenta a legibilidade do seu código. Assim como no operador OR, o tempo de execução aplica curto-circuito. Além de ser uma otimização de desempenho, você também pode usá-la em seu benefício ao trabalhar com valores nulos. O próximo exemplo, usa o operador AND para verificar se o argumento de entrada não é nulo e para executar um método nele. Se um curto-circuito não fosse usado nessa situação, o código lançaria uma exceção sempre que o parâmetro de entrada fosse nulo.

public void Process(string input)

{

bool result = (input != null) && (input.StartsWith(“v”));

// Do something with the result

}

O operador OR exclusivo (XOR) retorna verdadeiro somente quando exatamente um dos operandos é verdadeiro. Como o operador XOR precisa verificar se exatamente um dos operandos é verdadeiro, ele não aplica curto-circuito. A tabela abaixo fornece as possibilidades para o operador XOR.

|  |  |  |  |
| --- | --- | --- | --- |
| **Left operand** | **Right operand** | **Result** | **a = true E b = false** |
| True | True | False | a ^ a = False |
| True | False | True | a ^ b = True |
| False | True | True | b ^ a = True |
| False | False | False | b ^ b = False |

Para entender o porquê, é necessário que você saiba como o computador faz comparações. Para cada comparação feita, a CPU deve fazer o seguinte:

1. Busque a instrução e carregue-a na memória.
2. Incremente o ponteiro de instruções.
3. Visite a memória para obter o primeiro valor e armazene-o em um registro.
4. Acesse a memória para o segundo valor e armazene-o em um registro da CPU.
5. Faça a comparação e armazene o resultado em um registro da CPU.
6. Coloque a pilha do ponteiro de instruções para voltar ao local em que o código estava sendo executado antes a comparação.
7. Retorne o valor da comparação para o código.
8. Continue a execução na próxima instrução.

Para os computadores de hoje com CPUs rápidas, memória rápida, várias técnicas de cache e otimização de hardware, essas pequenas coisas podem parecer irrelevantes, mas um número suficiente delas combinadas pode ajudar a tornar seus programas mais eficientes.

**Tomando decisões**

Seu código pode executar tarefas simples sem a necessidade de decisões, mas em algum momento, seu código precisa avaliar uma condição e tomar uma ação apropriada com base no resultado dessa condição. Pode ser o resultado da entrada do usuário. Pode resultar do fato de um disco não estar na unidade ao ler ou gravar arquivos. Pode ser necessário verificar a presença de uma conexão de rede antes de enviar solicitações para um servidor.

À medida que você segue esses diferentes tipos de decisão a seguir, assegure-se de não apenas entender a sintaxe e como usá-las, mas também de entender por que um seria usado em detrimento de outro. O exame testa seu conhecimento de como implementá-los, mas a compreensão de quando usar uma estrutura de decisão específica pode ser útil em sua carreira como programador.

**A declaração if**

A declaração de controle de fluxo mais usada é a declaração if. A instrução if permite executar um trecho de código, dependendo de uma condição específica. A instrução a ser executada é executada apenas se a expressão booleana for avaliada como verdadeira. . O código abaixo mostra um exemplo de uso de if.

bool b = true;

if (b) Console.WriteLine(b);

Nesse caso, o aplicativo gera "True" porque a condição para a instrução if é verdadeira. Se b for falso, a instrução Console.WriteLine não será executada. É claro que passar um valor codificado para a instrução if não é muito útil. Normalmente, você usaria a instrução if com um valor mais dinâmico que pode mudar durante a execução do aplicativo. Ao trabalhar com instruções de fluxo de programa, é importante conhecer o conceito de um bloco de código, que permite escrever várias instruções em um contexto em que apenas uma única instrução é permitida. Todo o código no bloco é executado com base no resultado da instrução if. Você pode ver um exemplo disso abaixo.

bool b = true;

if (b)

{

int r = 42;

b = false;

}

As variáveis definidas em um bloco de código são acessíveis apenas dentro do bloco de código e ficam fora do escopo no final do bloco. Isso significa que você pode declarar variáveis dentro de um bloco e usá-las dentro do bloco, mas não fora dele. No exempo acima, a variável b é declarada fora do bloco e pode ser acessada no bloco externo e na instrução if. A variável r, no entanto, pode ser acessada apenas na instrução if.

Você também pode executar algum código quando a instrução if for avaliada como falsa. Você pode fazer isso usando um bloco else. A sintaxe geral é assim:

bool c = true;

if (b)

b = true;

else if (c)

c = true;

else

b = c = false;

Você também pode aninhar instruções if e else. Para facilitar a leitura, é bom delinear seu código corretamente. O código a seguir é perfeitamente legal, mas, à primeira vista, é difícil ver o que o código realmente faz:

if (b) b = true; else if (c) c = true; else b = c = false;

O compilador otimiza seu código e remove quaisquer chaves e declarações desnecessárias. Em circunstâncias normais, você deve se preocupar mais com a legibilidade do que com o número de linhas que você produz. Os membros da equipe agradecem especialmente quando você escreve um código que não apenas é correto, mas também mais fácil de manter.

**A instrução switch**

Você pode usar a instrução switch para simplificar instruções if complexas. Uma instrução switch verifica o valor de seu argumento e, em seguida, procura um rótulo correspondente. O exmplo abaixo mostra o código de uma instrução switch. A condição em uma instrução switch em idiomas anteriores, como C, tinha que ser do tipo int. C# permite comparar qualquer tipo de dados simples, como int, string, float e até mesmo enumerações.

void CheckWithSwitch(char input)

{

switch (input)

{

case 'a':

case 'e':

case 'i':

case 'o':

case 'u':

{

Console.WriteLine("Input is a vowel");

break;

}

case 'y':

{

Console.WriteLine("Input is sometimes a vowel.");

break;

}

default:

{

Console.WriteLine("Input is a consonant");

break;

}

}

}

Um switch pode usar uma ou várias seções de switch que podem conter um ou mais rótulos de switch, mas lembre-se de que nenhuma declaração de dois casos pode incluir o mesmo valor. Se desejar, também é possível adicionar um rótulo padrão usado quando nenhum dos outros rótulos corresponde. O ponto final de uma instrução switch não deve ser alcançável. Você precisa ter uma declaração, como break ou return, que sai explicitamente da instrução switch, ou precisa lançar uma exceção. Isso evita o comportamento fall-through que o C++ possui. Isso possibilita que as seções da opção apareçam em qualquer ordem sem afetar o comportamento. Em vez de cair implicitamente em outro rótulo, você pode usar a instrução goto.

int i = 1;

switch (i)

{

case 1:

{

Console.WriteLine("Case 1");

goto case 2;

}

case 2:

{

Console.WriteLine("Case 2");

break;

}

}

**O operador de coalescência nula**

O ?? O operador é chamado de operador de coalescência nula. Você pode usá-lo para fornecer um valor padrão para tipos de valor anuláveis ou para tipos de referência. O operador retornará o valor do lado esquerdo se não for nulo; caso contrário, o operando do lado direito. . O código abaixo mostra um exemplo de uso do operador.

int? x = null;

int? z = null;

int y = x ?? z ?? -1;

Obviamente, você pode conseguir o mesmo com uma instrução if, mas o operador de coalescência nula pode encurtar seu código e melhorar sua legibilidade.

**O operador condicional ternário**

O operador condicional (? :) retorna um dos dois valores, dependendo de uma expressão booleana. Se a expressão for verdadeira, o primeiro valor será retornado; caso contrário, o segundo. O código abaixo mostra um exemplo de como o operador pode ser usado para simplificar algum código. Nesse caso, a instrução if pode ser substituída pelo operador condicional.

int? valor = null;

if (true)

valor = 1;

else

valor = 0;

valor = p ? 1 : 0;

**ESTRUTURA REPETIÇÂO/LOOPS**

Outro assunto que tem a ver com o fluxo do seu programa é a iteração entre as coleções. As coleções são amplamente usadas em C#, e o idioma oferece construções que você pode usar com elas:

* **for**
* **foreach**
* **while**
* **do while**

**O loop for**

Você pode usar um loop for quando precisar percorrer uma coleção até que uma condição específica seja atingida (por exemplo, você atingiu o final de uma coleção). O exmplo abaixo mostra um exemplo no qual você percorre todos os itens em uma matriz.

int[] values = { 1, 2, 3, 4, 5, 6 };

for (int index = 0; index < values.Length; index++)

{

Console.Write(values[index]);

}

Como você pode ver, o loop for consiste em três partes diferentes:

**for (inicial; condição; loop)**

1. A parte inicial é executada antes da primeira iteração e declara e inicializa as variáveis ​​do loop.
2. A condição é avaliada em cada iteração. Quando a condição é igual a false, o loop é encerrado.
3. A seção de loop é executada durante todas as iterações e normalmente é usada para alterar o contador usado para fazer loop na coleção.

Você pode usar qualquer nome de variável para inicializar um loop for, mas lembre-se do seguinte:

* Você não pode usar palavras-chave para nomes de variáveis.
* A variável declarada não pode ter o mesmo nome que uma variável que você usa para outra finalidade no loop for.
* A variável usada como inicializador pode ser usada dentro do loop for.
* Você não pode usar essa variável fora do loop for devido ao escopo da variável.

Nenhuma dessas peças é necessária. Você pode usar for (;;) {} como um loop for perfeitamente legal que nunca terminaria. Você também pode usar várias instruções em cada parte do loop for.

for (int x = 0, y = values.Length - 1;

((x < values.Length) && (y >= 0));

x++, y--)

{

Console.Write(values[x]);

Console.Write(values[y]);

}

Também não é necessário permitir que o valor do loop aumente ou diminua com 1. Por exemplo, você pode aumentar o índice com 2 para exibir apenas os números ímpares, como mostra abaixo.

for (int index = 0; index < values.Length; index += 2)

{

Console.Write(values[index]);

}

Normalmente, o loop for termina quando a condição se torna falsa, mas você também pode decidir sair manualmente do loop. Você pode fazer isso usando a instrução break ou return quando desejar sair completamente do método. A seguir é mostrado um exemplo da instrução break.

for (int index = 0; index < values.Length; index++)

{

if (values[index] == 4) break;

Console.Write(values[index]);

}

Além de interromper completamente o loop, você também pode instruir o loop for a continuar para o próximo item usando a instrução continue. O exemplo a seguir mostra que o número 4 é ignorado no loop.

for (int index = 0; index < values.Length; index++)

{

if (values[index] == 4) continue;

Console.Write(values[index]);

}

Até agora, você viu apenas o iterador de loop for como contagem. Você pode usar qualquer um dos operadores de incremento C # nesta parte do loop for, o que significa que você pode aumentar ou diminuir (para diminuir em um determinado valor). Os seguintes operadores são aceitos para uso na seção do iterador de loop for:

|  |  |
| --- | --- |
| **Operador** | **Descrição** |
| ++ | Operador de incremento em que os valores são incrementados em um. |
| -- | Operador de decremento em que os valores são decrementados por um. |
| += | Operador que pode ser usado com literais para alterar a etapa, como += 2, que aumenta por um valor de 2 cada vez. |
| -= | Operador decremento do operador acima. |
| \*= | Operador incremento por um fator de multiplicação. |
| /= | Operador decréscimo por um fator de divisão. |

Uma consideração ao criar loops é que seu loop não precisa fazer nada. Um bloco de instruções vazio significa que nenhum código é executado durante o loop. O loop simplesmente itera até que a condição seja verdadeira:

// empty for loop

for (int counter = 0; counter >= 10; counter++)

{

;

}

O C# permite criar também um loop infinito, simplesmente criando o loop for sem nenhum dos valores entre parênteses. Você pode optar por usar um loop infinito em aplicativos em tempo real em que deseja uma pesquisa contínua de entradas ou talvez queira aplicar um teste de estresse a um aplicativo ou servidor. Apenas garanta que você é um meio de sair do loop, que às vezes é simplesmente fechar o aplicativo. Um loop sem escopo de saída, pode bloquear rapidamente um computador consumindo recursos de memória e CPU. Aqui está um exemplo em que não há inicializador, condição ou incremento:

// infinite for loop in C#

for (; ; )

{

;

}

**O loop while e do-while**

Outra construção de loop é o loop while. Um loop for nada mais é do que uma maneira conveniente de escrever um loop while que faz a verificação e o incremento do contador. O código abaixo mostra um exemplo. Observe os parênteses extras para restringir o escopo da variável de loop.

int[] values = { 1, 2, 3, 4, 5, 6 };

int index = 0;

while (index < values.Length)

{

Console.Write(values[index]);

index++;

}

Como você pode ver, um loop while verifica uma expressão e é executada enquanto essa expressão for verdadeira. Você deve usar um loop for quando souber o número de iterações com antecedência. Um loop while pode ser usado quando você não sabe o número de iterações.

Se a condição do loop while for falsa, ele não executará o código dentro do loop. Isso é diferente ao usar um loop do-while. Um loop do-while executa pelo menos uma vez, mesmo se a expressão for falsa. O código abaixo mostra um exemplo de uso de um loop do-while.

do

{

Console.WriteLine("Executed once!");

}

while (false);

Dentro de um loop while ou do-while, você pode usar as instruções continue e break, assim como no loop for. Pode haver muitos motivos para escolher um do-while over invés do while, mas um cenário típico é quando você espera a entrada do usuário e precisa garantir que a entrada seja recebida no loop em vez de fora do loop. Um exemplo ajuda a demonstrar:

char someValue;

do

{

someValue = (char)Console.Read();

Console.WriteLine(someValue);

} while (someValue != 'q');

**O loop foreach**

O loop foreach é usado para iterar sobre uma coleção e automaticamente armazena o item atual em uma variável de loop. O loop foreach monitora onde está a coleção e protege você contra a iteração após o final da coleção. O C# fornece a instrução foreach para iterar com coleções de itens em que a quantidade não é conhecida no tempo de execução, como alocações dinâmicas com base na entrada do usuário. Por exemplo, você pode criar uma matriz de caracteres a partir dos caracteres individuais de uma sequência de texto inserida por um usuário em tempo de execução. Outras possibilidades podem ser um conjunto de dados criado após o acesso a um banco de dados. Nos dois casos, você não saberá o número de valores no momento em que escreve o código. O código abaico mostra um exemplo de como usar o loop foreach.

int[] values = { 1, 2, 3, 4, 5, 6 };

foreach (int i in values)

{

Console.Write(i);

}

Como você pode ver, o loop foreach armazena automaticamente o item atual em uma variável fortemente tipada. Você pode usar as instruções continue e interromper para influenciar o funcionamento do loop foreach. Coleções são tipicamente matrizes, mas também outros objetos .NET que implementaram as interfaces IEnumerable. A variável de loop não pode ser modificada. Você pode fazer modificações no objeto para o qual a variável aponta, mas não pode atribuir um novo valor a ele. O código a seguir mostra essas diferenças.

class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

}

static void CannotChangeForeachIterationVariable()

{

var people = new List<Person>

{

new Person() { FirstName ="John", LastName ="Doe"},

new Person() { FirstName ="Jane", LastName = "Doe"},

};

foreach (Person p in people)

{

p.LastName = "Changed"; // This is allowed

//p = new Person(); // This gives a compile error

}

}

Você pode entender esse comportamento quando souber como o foreach realmente funciona. Quando o compilador encontra uma instrução foreach, ele gera algum código em seu nome; foreach é um açúcar sintático que permite escrever um código de maneira agradável. O exemplo abaixo mostra o que está acontecendo.

var people = new List<Person>{

new Person() { FirstName ="John", LastName ="Doe"},

new Person() { FirstName ="Jane", LastName = "Doe"}};

List<Person>.Enumerator e = people.GetEnumerator();

try

{

Person v;

while (e.MoveNext())

{

v = e.Current;

}

}

finally

{

System.IDisposable d = e as System.IDisposable;

if (d != null) d.Dispose();

}

Se você alterar o valor de e.Current para outra coisa, o padrão do iterador não pode determinar o que fazer quando o e.MoveNext é chamado. É por isso que não é permitido alterar o valor da variável de iteração em uma instrução foreach.

**ESTRUTURA SALTOS/JUMPS**

Outro tipo de declaração que pode ser usada para influenciar o fluxo do programa é uma declaração jump. As instruções de salto permitem que os controles do programa se movam de um ponto para outro em qualquer local específico durante a execução de um programa. Abaixo estão as instruções de salto que podemos usar em C#:

* **Goto**
* **Break**
* **Continue**
* **Return**
* **Throw**

**Instrução Goto**

Uma instrução goto é uma instrução jump que transfere seus controles para uma instrução rotulada. A instrução goto requer que o rótulo identifique o local para onde o controle irá. Um rótulo é qualquer identificador válido e deve ser seguido por dois pontos. O rótulo é colocado antes da declaração para onde o controle deve ser transferido. Se o rótulo não puder ser encontrado ou não estiver dentro do escopo da instrução goto, ocorrerá um erro do compilador. O código a seguir mostra um exemplo de uso de goto e um rótulo.

int x = 3;

if (x == 3) goto customLabel;

x++;

customLabel:

Console.WriteLine();

Console.WriteLine(x);

Você não pode pular para um rótulo que não está no escopo. Isso significa que você não pode transferir o controle para outro bloco de código que está fora do seu bloco atual. O compilador também garante que quaisquer blocos finally que intervenham sejam executados.

**Instrução Break**

Break é uma palavra-chave que também é uma instrução de salto, que finaliza o fluxo do programa em loop ou na instrução switch (ou seja, ignora o bloco atual e passa para o bloco ou código externo, se houver).

int[] numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

for (int i = 0; i < 10; i++)

{

if (numbers[i] == 3)

{

break;

}

Console.Write(numbers[i]);

}

Console.WriteLine("End of Loop");

Console.ReadLine();

Quando o snippet de código acima for executado, a saída será "End of Loop". Vamos entender como. Quando uma condição if escrita dentro do loop for satisfeita, a palavra-chave break será executada. Ele finaliza a iteração restante do loop e pula o controle para fora do loop e começará a executar o código gravado fora do loop, ou seja, "Console.WriteLine (" End of Loop ");"

**Instrução Continue**

A instrução Continue também é uma instrução de salto, que ignora a iteração atual e move o controle para a próxima iteração do loop. Continuar é uma palavra-chave, o mesmo que quebra, mas com o comportamento acima mencionado

int[] numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

for (int i = 0; i < 10; i++)

{

if (numbers[i] == 5)

{

continue;

}

Console.Write(numbers[i]);

}

Console.WriteLine("End of Loop");

Neste exemplo, o loop for funcionará normalmente como funciona, mas quando os números de valor [i] se tornarem 5, ele ignorará a iteração, o que significa que interromperá a execução da iteração atual e passará para a próxima iteração.

**Instrução Return**

Return também é uma instrução de salto, que retorna o controle do programa para o método de chamada. Ele retorna um valor ou nada, dependendo da natureza do método (ou seja, tipo de retorno do método). Return também é uma palavra-chave com o comportamento acima mencionado.

static int getAge()

{

return 20;

}

Console.WriteLine("Welcome to Exam 70-483 Certification");

int age = getAge();

Console.WriteLine("Age is: " + age);

Neste exemplo, o método getAge () é um tipo de int, portanto, o método retorna o valor do tipo int e o controle automaticamente vai para onde está chamando, ou seja, int age = getAge () no método principal. Portanto, o valor retornado pelo método getAge é armazenado na variável "age".

static void Main(string[] args)

{

Console.WriteLine("Welcome to Exam 70-483 Certification");

return;

Console.WriteLine("This Statement will never executed!");

Console.ReadLine();

}

No segundo exemplo acima, o método retornou o tipo void, o que não significa nada, portanto, não é necessário retornar valor. Nesse caso, usamos a instrução "return" sem um valor, o que ajuda a pular as instruções restantes do método e saltar o controle de volta para onde o método foi chamado.

Se a instrução return for usada no bloco try/catch e este try/catch finalmente bloquear, então o bloco finalmente será executado nessa condição também e após o controle retornar ao método de chamada. Código após a declaração de retorno está inacessível. Portanto, é aconselhável usar a declaração de retorno dentro do bloco if-else, se estivermos dispostos a pular a declaração restante do método somente quando uma determinada condição for atendida. Caso contrário, execute o método completo.

As instruções jump, como break e continue, podem ser usadas em algumas situações. Se possível, você deve evitá-los. Ao refatorar seu código, você pode removê-lo a maior parte do tempo e isso melhora a legibilidade do seu código. A declaração goto é ainda pior. É considerado uma má prática. Embora o C# restrinja a maneira como o operador goto se comporta, como uma diretriz, você deve evitar o uso de goto. Uma área em que o goto é usado está no código gerado, como o código que o compilador gera quando você usa o novo recurso de async/await no C# 5.

Sumário

* Expressões booleanas podem usar vários operadores: ==,! =, <,>, <=,> =,!. Esses operadores podem ser combinados usando AND (&&), OR (||) e XOR (^).
* Você pode usar a instrução if-else para executar o código, dependendo de uma condição específica.
* A instrução switch pode ser usada ao combinar um valor com algumas opções.
* O loop for pode ser usado ao iterar sobre uma coleção em que você conhece o número de iterações antecipadamente.
* Um loop while pode ser usado para executar algum código enquanto uma condição é verdadeira; do-while deve ser usado quando o código deve ser executado pelo menos uma vez.
* o foreach pode ser usado para iterar sobre coleções.
* Instruções jump como break, goto e continue podem ser usadas para transferir o controle para outra linha do programa.

Criar e implementar eventos e retornos de chamada

* Criar EventHandlers; assinar e cancelar assinatura de eventos (+= e =+); usar tipos delegados integrados (Func e Action) para criar eventos; criar delegados; expressões lambda (=>); métodos anônimos (delegate...)

Em qualquer linguagem moderna, o desenvolvimento orientado a eventos é usado para estruturar um programa em torno de vários eventos. Esses eventos executam uma certa funcionalidade quando uma determinada condição satisfaz, por exemplo, feche o aplicativo quando um usuário clicar no botão "Sair". Ou desligue o sistema quando a temperatura do calor aumentar, etc.

Um evento pode ser usado para fornecer notificações. Você pode se inscrever em um evento se estiver interessado nessas notificações. Você também pode criar seus próprios eventos e aumentá-los para fornecer notificações quando algo interessante acontecer. O .NET Framework oferece tipos internos que você pode usar para criar eventos. Usando delegados, expressões lambda e métodos anônimos, você pode criar e usar eventos de maneira confortável.

**DELEGATES**

Um delegate é um tipo que representa referências aos métodos com lista de parâmetros e tipo de retorno específicos. Um delegate é um tipo que encapsula com segurança um método, semelhante a um ponteiro de função em C e C++. No entanto, ao contrário dos ponteiros de função de C, delegates são orientados a objeto, fortemente tipados e seguros. Ao instanciar um delegate, você pode associar sua instância a qualquer método com assinatura e tipo de retorno compatíveis. Você pode invocar (ou chamar) o método através da instância do delegate.

Delegates são usados para passar métodos como argumentos a outros métodos. Os manipuladores de eventos nada mais são do que métodos chamados por meio de delegates. Ao criar um método personalizado, uma classe como um controle do Windows poderá chamá-lo quando um determinado evento ocorrer.

|  |  |
| --- | --- |
| **Versão** | **Sintaxe** |
| C# 1.0 | Introdução dos delegates |
| C# 2.0 | Oferece uma maneira mais simples |
| C# 2.0 e versões posteriores | Declarados e instanciados com método anônimo |
| C# 3.0 e versões posteriores | Declarados e instanciados com expressão lambda |

static void Notify(string name)

{

Console.WriteLine($"Notification received for: {name}");

}

delegate void \_delegado(string str);

static void Main(string[] args)

{

// No C# 1.0 e versões posteriores

\_delegado del1 = new \_delegado(Notify);

//O C# 2.0 oferece uma maneira mais simples

\_delegado del2 = Notify;

//No C# 2.0 e versões posteriores, com método anônimo

\_delegado del3 = delegate (string name)

{

Console.WriteLine($"Notification received for: {name}");

};

// No C# 3.0 e versões posteriores, com expressão lambda,

\_delegado del4 = name => { Console.WriteLine($"Notification received for: {name}"); };

}

Qualquer método de qualquer classe ou struct acessível que corresponda ao tipo delegate pode ser atribuído ao delegate. O método pode ser estático ou de instância. Isso possibilita alterar via programação chamadas de método e também conectar novo código a classes existentes.

Os delegates possuem as seguintes propriedades:

* São semelhantes a ponteiros de função do C++, mas delegates são totalmente orientados a objeto e, ao contrário dos ponteiros de C++ para funções de membro, os delegados encapsulam uma instância do objeto e um método.
* Permitem que métodos sejam passados como parâmetros.
* Podem ser usados para definir métodos de retorno de chamada.
* Podem ser encadeados juntos; por exemplo, vários métodos podem ser chamados um único evento.
* Os métodos não precisam corresponder ao tipo delegado exatamente.
* O C# versão 2.0 introduziu o conceito de [métodos anônimos](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/operators/delegate-operator) que permitem que blocos de código sejam passados como parâmetros em vez de um método definido separadamente.
* O C# 3.0 introduziu expressões lambda como uma maneira mais concisa de escrever blocos de código embutidos. Os métodos anônimos e as expressões lambda (em determinados contextos) são compilados para tipos delegados. Juntos, esses recursos são agora conhecidos como funções anônimas.

Um objeto delegate é normalmente construído fornecendo-se o nome do método que o delegate encapsulará ou como uma [função anônima](https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/statements-expressions-operators/anonymous-functions). Quando um delegado é instanciado, uma chamada de método feita ao delegate será passada pelo delegate para esse método. Os parâmetros passados para o delegate pelo chamador são passados para o método e o valor de retorno, se houver, do método é retornado ao chamador pelo delegate. Isso é conhecido como invocar o delegate. Um delegate instanciado pode ser invocado como se fosse o método encapsulado em si. Por exemplo:

delegate void del\_invoca(string str);

static void Main(string[] args)

{

// Instantiate the delegate.

del\_invoca handler = DelegateMethod;

// Call the delegate.

handler("Hello World");

}

// Create a method for a delegate.

public static void DelegateMethod(string message)

{

Console.WriteLine(message);

}

O delegado também pode ser chamado usando o método .invoke. Veja o seguinte trecho de código.

handler.Invoke("Ali Asad");

Tipos de delegate são derivados da classe [Delegate](https://docs.microsoft.com/pt-br/dotnet/api/system.delegate) do .NET Framework. Tipos de delegate são [lacrados](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/sealed) – não podem ser derivados de – e não é possível derivar classes personalizadas de [Delegate](https://docs.microsoft.com/pt-br/dotnet/api/system.delegate). Você pode armazenar valores delegados em variáveis, assim como qualquer outro tipo de valor. A única questão confusa é que os valores sendo manipuladas são referências a métodos, e não a algum tipo de dado mais concreto, como um int ou string. Por exemplo, você pode declarar que uma única variável é do tipo delegado; você pode criar uma estrutura que possua propriedades ou campos de um tipo de delegado; e você pode criar uma matriz de variáveis de um tipo de delegado. Você pode até criar uma lista de valores do tipo. Por exemplo, o código a seguir cria uma lista que pode conter referências a métodos que correspondem ao tipo FunctionDelegate.

public delegate int del\_Calculate(int x, int y);

List<del\_Calculate> function = new List<del\_Calculate>();

**Delegate as a Callback**

Como o delegate instanciado é um objeto, ele pode ser passado como um parâmetro ou atribuído a uma propriedade. Isso permite que um método aceite um delegate como um parâmetro e chame o delegate posteriormente. Isso é conhecido como um retorno de chamada assíncrono (Callback) e é um método comum de notificação de um chamador quando um processo longo for concluído.

Quando um delegate é usado dessa maneira, o código que usa o delegate não precisa de conhecimento algum da implementação do método que está sendo usado. A funcionalidade é semelhante ao encapsulamento que as interfaces fornecem.

Outro uso comum de chamadas de retorno é definir um método de comparação personalizada e passar esse delegate para um método de classificação. Ele permite que o código do chamador se torne parte do algoritmo de classificação. O método de exemplo a seguir usa um delegate como um parâmetro:

public delegate void del\_invoca(string message);

// Create a method for a delegate.

public static void DelegateMethod(string message)

{

Console.WriteLine(message);

}

static public void MethodWithCallback(int param1, int param2, del\_invoca callback)

{

callback("The number is: " + (param1 + param2).ToString());

}

Em seguida, você pode passar o delegado criado acima para esse método e receber a seguinte saída para o console:

static void Main(string[] args)

{

del\_invoca handler = DelegateMethod;

MethodWithCallback(1, 2, handler); //The number is: 3

}

Usando o delegate como uma abstração, MethodWithCallback não precisa chamar o console diretamente — ele não precisa ser criado com um console em mente. O que MethodWithCallback faz é simplesmente preparar uma cadeia de caracteres e passá-la para outro método. Isso é especialmente poderoso, uma vez que um método delegado pode usar qualquer número de parâmetros.

**Multicast('+' || '-='ou'-' || '-=').**

Quando um delegate é construído para encapsular um método de instância, o delegate faz referência à instância e ao método. Um delegate não tem conhecimento do tipo de instância além do método que ele encapsula, de modo que um delegate pode se referir a qualquer tipo de objeto desde que haja um método nesse objeto que corresponda à assinatura do delegate.

Quando um delegate é construído para encapsular um método estático, ele só faz referência ao método. Considere as seguintes classe MethodInstance:

public class MethodInstance

{

public void Method\_01(string message)

{

Console.WriteLine("Instance Método 01: " + message);

}

public void Method\_02(string message)

{

Console.WriteLine("Instance Método 02: " + message);

}

}

Com o método estático StaticMethod mostrado abaixo, temos três métodos diferentes que podem ser encapsulados por uma mesma instância delegate del\_assign.

delegate void del\_assign(string str);

static void Main(string[] args)

{

var met\_instance = new MethodInstance();

//Both types of assignment are valid.

del\_assign del\_01 = met\_instance.Method\_01;

del\_assign del\_02 = met\_instance.Method\_02;

del\_assign del\_03 = StaticMethod;

del\_assign allMethodsDelegate = del\_01 + del\_02;

allMethodsDelegate += del\_03;

allMethodsDelegate("MultiCast");

int invocationCount = del\_01.GetInvocationList().GetLength(0);

Console.WriteLine("Métodos em del\_01: " + invocationCount);

int allinvocationCount = allMethodsDelegate.GetInvocationList().GetLength(0);

Console.WriteLine("Métodos em allinvocationCount: " + allinvocationCount);

}

public static void StaticMethod(string message)

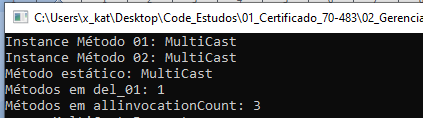
{

Console.WriteLine("Método estático: " + message);

}

Um delegate pode chamar mais de um método quando invocado. Isso é chamado de multicast. Para adicionar um método extra à lista de métodos do delegate basta adicionar os operadores de adição (+) ou de atribuição de adição (+=). Nesse ponto, allMethodsDelegate contém três métodos em sua lista de invocação — Method\_01, Method\_02 e StaticMethod. Os três delegates originais, del\_01, del\_02 e del\_03, permanecem inalterados. Quando allMethodsDelegate é invocado, os três métodos são chamados na ordem. Se o delegate usar parâmetros de referência, a referência será passada em sequência para cada um dos três métodos por vez, e quaisquer alterações em um método serão visíveis no próximo método.

allMethodsDelegate("MultiCast");



Se o delegate tiver um valor de retorno e/ou parâmetros de saída, ele retornará o valor de retorno e os parâmetros do último método invocado.

public static int SubtractRetorno(int a, int b)

{

var subtrai = a - b;

return subtrai;

}

delegate int del\_operacoes\_retorno(int x, int y);

static void Main(string[] args)

{

//Poderia voltar um retorno; delegate int

del\_operacoes\_retorno subt = SubtractRetorno;

var resultado = subt(4, 5);

Console.WriteLine("SubtraiNumbers: " + resultado); // SubtraiNumbers: -1

resultado = subt(10, 5);

resultado = subt(1, 100);

Console.WriteLine("SubtraiNumbers: " + resultado); // SubtraiNumbers: -99

Console.ReadKey();

}

Se você definir uma variável delegate igual a um método estático, ficará claro o que acontece quando você invoca o método da variável. Há apenas um método compartilhado por todas as instâncias da classe que a define, portanto esse é o método chamado. Se você definir uma variável delegate igual a um método de instância, os resultados serão um pouco mais confusos. Quando você invoca o método da variável, ele é executado na instância em que você usou para definir o valor da variável. Como no exemplo abaixo:

public class MethodInstance

{

public delegate void GetStringDelegate();

public static void StaticMethod\_01()

{

Console.WriteLine("Static Método");

}

public void InstanceMethod\_02()

{

Console.WriteLine("Instance Método Name:" + Name);

}

// Variables to hold GetStringDelegates.

public GetStringDelegate StaticMethod;

public GetStringDelegate InstanceMethod;

public GetStringDelegate PrintMethods;

}

MethodInstance alice = new MethodInstance() { Name = "Alice" };

MethodInstance bob = new MethodInstance() { Name = "Bob" };

//Static e Intance methods

alice.StaticMethod = MethodInstance.StaticMethod\_01;

alice.InstanceMethod = alice.InstanceMethod\_02;

bob.StaticMethod = MethodInstance.StaticMethod\_01;

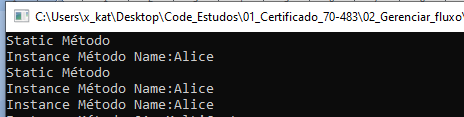
bob.InstanceMethod = alice.InstanceMethod\_02;

bob.PrintMethods = alice.StaticMethod + alice.InstanceMethod;

bob.PrintMethods += bob.StaticMethod + bob.InstanceMethod;

bob.PrintMethods += bob.InstanceMethod;

bob.PrintMethods();



Esse código cria dois objetos chamados alice e bob. A variável InstanceMethod do objeto bob se refere à método InstanceMethod da instância de Alice, portanto também retorna "Alice".

Quando algum dos métodos gerar uma exceção que não foi detectada dentro do método, essa exceção será passada ao chamador do delegate e nenhum método subsequente na lista de invocação será chamado. Para remover um método da lista de invocação, use os [operadores de atribuição de subtração ou subtração](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/operators/subtraction-operator) (- ou -=). Por exemplo:

allMethodsDelegate += del\_04;

//remove Method1

allMethodsDelegate -= del\_04;

// copy AllMethodsDelegate while removing d2

del\_assign oneMethodDelegate = allMethodsDelegate - del\_04;

Como os tipos de delegates são derivados de System.Delegate, os métodos e as propriedades definidos por essa classe podem ser chamados no delegate. Por exemplo, para localizar o número de métodos na lista de invocação do delegado, é possível escrever:

int allinvocationCount = allMethodsDelegate.GetInvocationList().GetLength(0);

Console.WriteLine("Métodos em allinvocationCount: " + allinvocationCount);

Delegates com mais de um método em sua lista de invocação derivam de [**MulticastDelegate**](https://docs.microsoft.com/pt-br/dotnet/api/system.multicastdelegate), que é uma subclasse de System.Delegate. O código acima funciona em ambos os casos, pois as classes oferecem suporte à GetInvocationList. É possível também fazer um loop sobre cada método usando o método GetInvocationList.

foreach (delegateName item in del.GetInvocationList())

{

//invoke each method, and display return value

Console.WriteLine(item());

}

Delegates multicast são amplamente usados na manipulação de eventos. Objetos de origem do evento enviam notificações de eventos aos objetos de destinatário que se registraram para receber esse evento. Para se registrar para um evento, o destinatário cria um método projetado para lidar com o evento, em seguida, cria um delegate para esse método e passa o delegate para a origem do evento. A origem chama o delegate quando o evento ocorre. O delegate chama então o método de manipulação de eventos no destinatário, fornecendo os dados do evento. O tipo de delegate de um determinado evento é definido pela origem do evento.

A comparação de delegates de dois tipos diferentes atribuídos no tempo de compilação resultará em um erro de compilação. Se as instâncias de delegate forem estaticamente do tipo System.Delegate, então a comparação será permitida, mas retornará false no tempo de execução. Por exemplo:

delegate void Delegate1();

delegate void Delegate2();

static void CompareDelegates(Delegate1 d, Delegate2 e, System.Delegate f)

{

// Compile-time error. Se não colocar o Cast

Console.WriteLine(d == (System.Delegate)e); //False

// OK at compile-time. False if the run-time type of f

// is not the same as that of d.

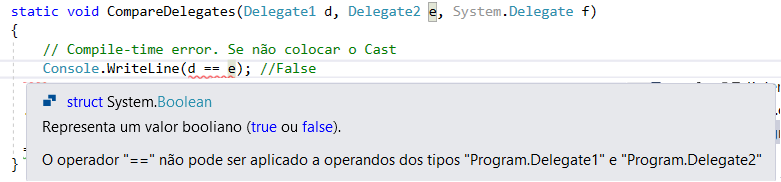
Console.WriteLine(e == f); //True

}

Delegate1 del01 = StaticMethod;

Delegate2 del02 = StaticMethod;

CompareDelegates(del01, del02, del02);



**Covariância e contravariância**

Com variância nos delegates, o método não precisa corresponder ao tipo de delegado. Como a variância fornece um grau de flexibilidade ao combinar um tipo de delegado com a assinatura do método, podemos usar a variância das duas maneiras a seguir.

1. **Covariância**: A covariância é aplicada no tipo de retorno de um método. A covariância torna possível que um método tenha um tipo de retorno mais derivado do que o definido no delegado
2. **Contravariância:** A contravariância é aplicada ao tipo de parâmetro de um método. A contravariância permite um método que possui tipos de parâmetros menos derivados do que aqueles no tipo de delegado.

No C#, a covariância e a contravariância habilitam a conversão de referência implícita para tipos de matriz, tipos de delegados e argumentos de tipo genérico. A covariância preserva a compatibilidade de atribuição, e a contravariância reverte. O código a seguir demonstra a compatibilidade da atribuição entre tipos:

string str = "teste";

object obj = str;

Console.WriteLine("obj=" + obj); //obj = teste

Console.WriteLine("str=" + str); //str = teste

IEnumerable<string> strings = new List<string>();

IEnumerable<object> objetos = strings;

// strings=System.Collections.Generic.List`1[System.String]

Console.WriteLine("strings=" + strings.GetType());

// objetos = System.Collections.Generic.List`1[System.String].

Console.WriteLine("objetos=" + objetos.GetType());

Um objeto de um tipo mais derivado (string) é atribuído a um objeto de um tipo menos derivado (object). A compatibilidade da atribuição é preservada. Um exemplo de contravariância pode ser visto abaixo.

static void SetObject(object obj) { }

Action<object> actObject = SetObject;

Action<string> actString = actObject;

// actObject = System.Action`1[System.Object]

Console.WriteLine("actObject=" + actObject.GetType());

// actString = System.Action`1[System.Object]

Console.WriteLine("actString=" + actString.GetType());

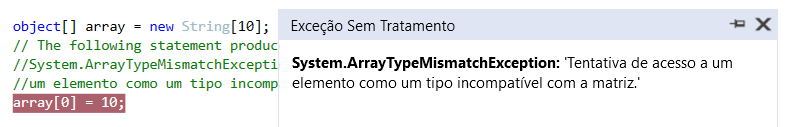
Aqui, um objeto que é instanciado com um argumento de tipo menos derivado é atribuído a um objeto instanciado com um argumento de tipo mais derivado. A compatibilidade da atribuição é revertida

A covariância para matrizes permite a conversão implícita de uma matriz de um tipo mais derivado para uma matriz de um tipo menos derivado. Mas essa operação não é fortemente tipada, conforme mostrado no exemplo de código a seguir.

object[] array = new String[10];

// The following statement produces a run-time exception.

array[0] = 10;



O exemplo de covariança abaixo mostra a utlização de classes do System.IO e uso de um delegate não genéricos.

public delegate TextWriter CovarianceDel();

static void Main(string[] args)

{

CovarianceDel del;

del = MethodStream;

del = MethodString;

}

public static StreamWriter MethodStream() { return null; }

public static StringWriter MethodString() { return null; }

Como o StreamWriter e o StringWriter herdam do TextWriter, você pode usar o delegate CovarianceDel com os dois métodos. Um exemplo de contravariância pode ser visto abaixo.

public delegate void ContravarianceDel(StreamWriter tw);

static void Main(string[] args)

{

ContravarianceDel del = DoSomething;

}

static void DoSomething(TextWriter tw) { }

Como o método DoSomething pode funcionar com um TextWriter, certamente também pode funcionar com um StreamWriter. Por causa da contravariância, você pode chamar o delegado e passar uma instância do StreamWriter para o método DoSomething. A seguir um exemplo de covariância e contravariância para um hierarquia de classes e delegados não genéricos.

class Parent { }

class Child : Parent { }

delegate Parent CovarianceHandle();

delegate void ContravarianceHandle(Child c);

static void Main(string[] args)

{

CovarianceHandle covarianca = CovarianceMethod;

covarianca();

ContravarianceHandle contra\_varianca = ContravarianceMethod;

Child child = new Child();

contra\_varianca(child);

Console.ReadKey();

}

static Child CovarianceMethod()

{

Console.WriteLine("Covariance Method");

return new Child();

}

static void ContravarianceMethod(Parent p)

{

Child ch = p as Child;

Console.WriteLine("Contravariance Method");

}

No .NET Framework 4 ou mais recente, o C# dá suporte à covariância e á contravariância em interfaces e delegados genéricos e permite a conversão implícita de parâmetros de tipo genérico. Uma interface ou delegado genérico será chamado variante se seus parâmetros genéricos forem declarados covariantes ou contravariantes. O C# permite que você crie suas próprias interfaces variantes e delegados.

O exemplo a seguir ilustra os benefícios do suporte à covariância nos delegados genéricos Func. O método FindByTitle assume um parâmetro do tipo String e retorna um objeto do tipo Employee. No entanto, você pode atribuir esse método ao delegado Func<String, Person> porque Employee herda Person.

// Hierarquia simples de classes.

public class Person { }

public class Employee : Person { }

static Employee FindByTitle(String title)

{

// This is a stub for a method that returns

// an employee that has the specified title.

return new Employee();

}

static void Func\_Covarianca()

{

// Crie uma instância do delegado sem usar variação.

Func<String, Employee> findEmployee = FindByTitle;

// O delegado espera que um método retorne Person,

// mas você pode atribuir a ele um método que retorne Employee.

Func<String, Person> findPerson = FindByTitle;

// Você também pode atribuir um delegado retorna um tipo mais derivado

// para um delegado que retorna um tipo menos derivado.

findPerson = findEmployee;

}

O exemplo a seguir ilustra os benefícios do suporte à contravariância nos delegados genéricos Action. O método AddToContacts assume um parâmetro do tipo Person. No entanto, você pode atribuir esse método ao delegado Action<Employee> porque Employee herda Person.

// Hierarquia simples de classes.

public class Person { }

public class Employee : Person { }

static void AddToContacts(Person person)

{

// This method adds a Person object

// to a contact list.

}

static void Action\_Contravarianca()

{

// Crie uma instância do delegado sem usar variação.

Action<Person> addPersonToContacts = AddToContacts;

// O delegado da ação espera

// um método que possui um parâmetro Employee,

// mas você pode atribuir a ele um método que possui um parâmetro Person

// porque Employee deriva de Person.

Action<Employee> addEmployeeToContacts = AddToContacts;

// Você também pode atribuir um delegado

// que aceita um parâmetro menos derivado para um delegado

// que aceita um parâmetro mais derivado.

addEmployeeToContacts = addPersonToContacts;

}

**Método Anônimo**

Um método anônimo é um método sem nome. Estes são métodos que são definidos com uma palavra-chave delegate. Um método anônimo não possui um tipo de retorno em sua assinatura. Seu tipo de retorno depende do tipo de variável delegate que mantém sua referência.

//Anonymous method that doesn't return value

Action act = delegate ()

{

Console.WriteLine("Inside Anonymous method");

};

//Anonymous method that does return value

Func<int, int> func = delegate (int num)

{

Console.Write("Inside Func: ");

return (num \* 2);

};

act();

Console.WriteLine(func(4)); // Inside Func: 8

Abaixo um exemplo de como passar um método anônimo como argumento de método.

static void Main(string[] args)

{

TestAnonymous(delegate ()

{

Console.WriteLine("Pass anonymous method in method's perameter");

});

}

public static void TestAnonymous(Action act)

{

act();

}

O código a seguir armazena um método anônimo em uma variável de um tipo de delegado.

private static Func<float, float> Function = delegate (float x) { return x \* x; };

var resultado = Function(Convert.ToSingle(4.3));

Console.WriteLine("Quadrado:" + resultado ); // Quadrado:18,49

Esse código declara uma variável denominada Function do tipo definido pelo delegado interno do Func que usa float como parâmetro e retorna um float. Ele define a variável Function igual a um método que retorna seu parâmetro ao quadrado. O programa não pode se referir a esse método pelo nome porque é anônimo, mas pode usar a variável Function para invocar o método

A linha de código anterior mostra como você pode fazer uma variável delegada se referir a um método anônimo. Dois outros lugares em que os programadores costumam usar métodos anônimos estão definindo manipuladores de eventos simples e executando tarefas simples em threads separados. O código a seguir adiciona um manipulador de eventos ao evento Paint de um formulário

public Form1()

{

InitializeComponent();

this.Paint += delegate (object obj, PaintEventArgs args)

{

args.Graphics.DrawEllipse(Pens.Red, 10, 10, 200, 100);

};

}

Quando o formulário recebe um evento Paint, o método anônimo desenha uma elipse vermelha. O código a seguir executa um método anônimo em um thread separado:

Thread thread = new Thread(delegate ()

{

Console.WriteLine("Hello World");

});

Thread.Sleep(3000);

thread.Start();

Esse código cria um novo objeto Thread, passando uma referência ao método anônimo. Quando o Thread é iniciado, ele executa esse método, neste caso exibindo uma mensagem.

**Expressão Lambda**

A expressão lambda é uma versão melhor da implementação do método anônimo. Para criar uma expressão lambda, especificamos parâmetros de entrada (se houver) no lado esquerdo do operador lambda => e colocamos o bloco de expressão ou instrução no outro lado.

Os exemplos aqui armazenam expressões lambda em variáveis delegadas porque são fáceis de descrever dessa maneira. Em muitos aplicativos, as expressões lambda são adicionadas à lista de manipuladores de eventos, passadas para métodos que tomam delegados como parâmetros ou são usados em expressões LINQ.

//Lambda Expression that doesn't return value

Action act = () =>

{

Console.WriteLine("Inside Lambda Expression");

};

//Lambda Expression that does have return value

Func<int, int> func = (int num) =>

{

Console.Write("Inside Func: ");

return (num \* 2);

};

act();

Console.WriteLine(func(4)); // Inside Func: 8

Se o corpo de um método anônimo contiver apenas uma única declaração, mencione os chavetas "{}" e uma palavra-chave de retorno com o valor retornado é opcional. Veja o seguinte snippet de código

//Lambda Expression that doesn't return value

Action actinline = () => Console.WriteLine("Hello World");

//Lambda Expression that does have return value

Func<int, int> funcinline = (int num) => num \* 2;

actinline();

Console.WriteLine(funcinline(4));

A expressão lambda também oferece a capacidade de não especificar um tipo de parâmetro. Seu tipo de parâmetro dependerá do tipo de parâmetro do tipo delegado que mantém sua referência. Veja o seguinte trecho de código.

//type of name will be string

Action<string> actName = (name) => Console.WriteLine(name);

//for single parameter, we can neglect () paranthese

Action<string> actName2 = name => Console.WriteLine(name);

Func<int, int> mul = x => x \* 2;

actName("Hello");

actName2("World");

Console.WriteLine(mul(4)); // 8

Passar expressão lambda em um parâmetro de método

static void TestLambda(Action act)

{

Console.WriteLine("Test Lambda Method");

act();

}

TestLambda(() => Console.WriteLine("Inside Lambda"));

Basicamente, você pode usar a palavra-chave async para indicar que um método pode ser executado de forma assíncrona. Você pode usar a palavra-chave wait para fazer com que um trecho de código chame um método assíncrono e aguarde o retorno. Geralmente, um método assíncrono é nomeado, mas você pode usar a palavra-chave async para tornar as expressões lambda assíncronas também.

private int Trials = 0;

public Form1()

{

InitializeComponent();

button1.Click += async (button, buttonArgs) =>

{

int trial = ++Trials;

label1.Text = "Running trial " + trial.ToString() + "...";

await DoSomethingAsync();

label1.Text = "Done with trial " + trial.ToString();

};

}

// Do something time consuming.

async Task DoSomethingAsync()

{

// In this example, just waste some time.

await Task.Delay(5000);

}

**DELEGATES GENÉRICOS INTERNOS**

O C# 3.0 inclui Func, Action e Predicate, que são delegates genéricos internos no namespace System. Esses tipos internos fornecem uma notação abreviada que praticamente elimina a necessidade de declarar a todo momento um delegate

**Func**

O Func possui de zero a 16 parâmetros de entrada e um parâmetro de saída. O último parâmetro é considerado como um parâmetro de saída.Um delegate Func com dois parâmetros de entrada e um parâmetro de saída será representado como abaixo.

// Hierarquia simples de classes.

public class Person { }

public class Employee : Person { }

public delegate TResult del\_func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);

private static int PersonParameter2(Employee employ, Person person) { return 10; }

del\_func<Employee, Person, int> del\_03 = PersonParameter2;

* Func é um delegate interno no namespace System.
* O Func deve retornar um valor. = delegate
* O Func pode ter de zero a 16 parâmetros de entrada.
* O Func não permite parâmetros ref e out.
* O Func pode ser usado com um método anônimo ou expressão lambda.

static void Main(string[] args)

{

Func<int, int, int> subtrai\_func = SubtraiNumbers;

var resultado = subtrai\_func(10, 5);

Console.WriteLine("Func SubtraiNumbers: " + resultado);

}

public static int SubtraiNumbers(int a, int b)

{

var subtrai = a - b;

return subtrai;

}

O delegate Func deve incluir um parâmetro out para resultado. Por exemplo, o seguinte método anônimo ao delegate Func não possui nenhum parâmetro de entrada, ele inclui apenas um parâmetro de saída.

Func<int> getRandomNumber = delegate ()

{

Random rnd = new Random();

var numero = rnd.Next(1, 100);

return numero;

}; };

Console.WriteLine("Func Anonima Random de 100: " + getRandomNumber());

O delegate Func também pode ser usado com uma expressão lambda, como mostrado abaixo:

getRandomNumber = () => new Random().Next(1, 100);

Func<int, int, int> Sum = (x, y) => x + y;

Console.WriteLine("Func Lambda Random de 100: " + getRandomNumber());

Console.WriteLine("Func Lambda Soma: " + Sum(5, 300));

Exemplos de declaração de um delegado Func:

public Func<string, bool> OnChangeFunc;

public Func<string, bool> OnChangeFunc2 { get; set; }

public Func<string, bool> OnChangeFunc = delegate (string str) { return true; };

**Action**

O delegado Action genérico representa um método que retorna nulo. Versões diferentes do Action levam entre 0 e 18 parâmetros de entrada. O código a seguir mostra a definição do delegado Action que usa dois parâmetros:

// Hierarquia simples de classes.

public class Person { }

public class Employee : Person { }

public delegate void del\_action<in T1, in T2>(T1 arg1, T2 arg2);

private static void PersonParameter2(Employee employ, Person person) { }

del\_action<Employee, Person> del\_03 = PersonParameter2;

A palavra-chave “in” da lista de parâmetros genéricos indica que os parâmetros dos tipos T1 e T2 são contravariantes. A menos que você precise definir um delegado com mais de 18 parâmetros, você pode usar Action em vez de criar seus próprios delegados. Por exemplo, o código abaixo que define um tipo EmployeeParameterDelegate que usa um objeto Employee como parâmetro e retorna nulo.

private delegate void EmployeeParameterDelegate(Employee employee);

private static EmployeeParameterDelegate EmployeeParameterMethod;

A primeira instrução deste código define o delegado EmployeeParameterDelegate. A declaração seguinte declara uma variável desse tipo. Estas intruções poderiam ser resumidas em uma linha declara uma variável comparável do tipo Action<Employee>.

private static Action<Employee> EmployeeParameterMethod2;

// A method that takes a Person as a parameter.

private static void PersonParameter(Person person) { }

// Use contravariance to set EmployeeParameterMethod = PersonParameter.

EmployeeParameterMethod = PersonParameter;

EmployeeParameterMethod2 = PersonParameter;

Um Action delegate é igual ao delegate Func, exceto que o Action delegate não retorna um valor. Em outras palavras, um Action delegate pode ser usado com um método que possui um tipo de retorno nulo.

* No Action o tipo de retorno deve ser nulo.
* O Action pode ter de 0 a 16 parâmetros de entrada.
* O Action pode ser usado com métodos anônimos ou expressões lambda.

public static void AddNumbers(string funcao, int a, int b)

{

var soma = a + b;

Console.WriteLine(funcao + " AddNumbers: " + soma);

}

static void Main(string[] args)

{

Action<string, int, int> soma\_action = AddNumbers;

soma\_action("Action", 100, 5);

}

Você pode inicializar um delegado Action usando a palavra-chave new ou atribuindo diretamente um método:

Action<string, int, int> soma\_action = AddNumbers;

Action<string, int, int> soma\_action2 = new Action<string, int, int>(AddNumbers);

Um método anônimo também um expressão Lambda pode ser atribuído a um Action delegate, por exemplo:

Action<int> AnonimaActionDel = delegate (int i)

{

Console.WriteLine("Action Anonima Numero: " + i);

};

AnonimaActionDel(10);

Action<int> LambdaActionDel = i => Console.WriteLine("Action Lambda Numero: " + i);

LambdaActionDel(10);

Exemplos de declaração de um delegado Action:

public Action OnChange { get; set; }

public Action<string> OnChangeParam;

public event Action OnChange = delegate { };

public Action<string> OnChangeParam = delegate (string str) { };

**Predicate**

Um predicado também é um delegate, como os delegates Func e Action. Representa um método que contém um conjunto de critérios e verifica se o parâmetro passado atende a esses critérios ou não(retornar um booleano - verdadeiro ou falso).

* No Predicate o tipo de retorno deve ser um booleano.
* O Predicate pode ter de 0 a 16 parâmetros de entrada.
* O Predicate pode ser usado com métodos anônimos ou expressões lambda.

public static bool IsUpperCase(string str)

{

return str.Equals(str.ToUpper());

}

Predicate<string> Predicate\_isUpper = IsUpperCase;

bool result = Predicate\_isUpper("hello world!!");

Console.WriteLine("Predicate: " + result); // Predicate: False

result = Predicate\_isUpper("HELLO!");

Console.WriteLine("Predicate: " + result); // Predicate: True

Um método anônimo e expressão lambda também pode ser atribuído a um tipo de delegate do Predicate, como mostrado abaixo.

Predicate<string> AnonimaPredicate = delegate (string s) { return s.Equals(s.ToUpper()); };

// Anonima Predicate: False

Console.WriteLine("Anonima Predicate: " + AnonimaPredicate("olá mundo !!"));

Predicate<string> LambdaPredicate = s => s.Equals(s.ToUpper());

// Lambda Predicate: False

Console.WriteLine("Lambda Predicate: " + LambdaPredicate("olá mundo !!"));

**Converter**

Representa um método que converte um objeto de um tipo para outro tipo.

public delegate TOutput del\_converter<in TInput, out TOutput>(TInput input);

* **TInput**>> O tipo de objeto que deve ser convertido. Este parâmetro de tipo é contravariante. Ou seja, você pode usar o tipo que você especificou ou qualquer tipo que seja menos derivado.
* **TOutput**>> O tipo para o qual o objeto de entrada deve ser convertido. Este parâmetro de tipo é covariante. Ou seja, você pode usar o tipo especificado ou qualquer tipo mais derivado.
* **TInput input**>> método que realiza a conversão.

public static string ConvertUsuario\_ToString(Usuario user)

{

return user.Nome;

}

del\_converter<Usuario, string> del\_03 = ConvertUsuario\_ToString;

var romeu = new Usuario("Romeu", 10);

var nome = del\_03(romeu);

Console.WriteLine(nome);

A seguir dois exemplos de código. O primeiro demonstra o delegado Converter <TInput, TOutput> com o método ConvertAll da classe Array e o segundo demonstra o delegado com o método ConvertAll da classe genérica List <T> .

public class Usuario

{

public string Nome { get; set; }

public int Idade { get; set; }

public Usuario(string nome, int idade)

{

this.Nome = nome;

this.Idade = idade;

}

}

class Program

{

public delegate TOutput del\_converter<in TInput, out TOutput>(TInput input);

static void Main(string[] args)

{

del\_converter<Usuario, string> del\_03 = ConvertUsuario\_ToString;

var romeu = new Usuario("Romeu", 10);

var nome = del\_03(romeu);

Console.WriteLine(nome);

Console.WriteLine("=========FloatOriginal============");

// Crie uma matriz de objetos PointF.

PointF[] apf = { new PointF (27.8F, 32.62F),

new PointF (99.3F, 147.273F), new PointF (7.5F, 1412.2F)};

// Exibe cada elemento na matriz PointF.

foreach (PointF p in apf) Console.WriteLine(p);

// Converte cada elemento PointF em um objeto Point.

// Repare que é utilizado a classe Array com o ConvertAll

Point[] ap = Array.ConvertAll(apf,

new Converter<PointF, Point>(Array\_PointFToPoint));

// Exibe cada elemento na matriz Point.

Console.WriteLine();

Console.WriteLine("======Converter Float Array to int======");

foreach (Point p in ap) Console.WriteLine(p);

Console.WriteLine();

Console.WriteLine("======Converter Usuarios List to string======");

List<Usuario> users = new List<Usuario>()

{

new Usuario("Romeu", 10),

new Usuario("Julieta", 90),

new Usuario("Hamlet", 55)

};

// Repare que é utilizado o objeto users com o ConvertAll

List<string> nomes = users.ConvertAll<string>(new Converter<Usuario, string>(ConvertUsuario\_ToString));

foreach (var p in nomes) Console.WriteLine(p);

Console.ReadKey();

}

public static Point Array\_PointFToPoint(PointF pf)

{

return new Point(((int)pf.X), ((int)pf.Y));

}

public static string ConvertUsuario\_ToString(Usuario user)

{

return user.Nome;

}

}

**Comparison<T> Delegate**

Esse tipo permite a classificação personalizada.É frequentemente usado com Array.Sort ou List.Sort. Implementamos Comparison (T) usando seu construtor. Ele é um método que recebe 2 parâmetros e retorna um int.

public delegate int del\_comparison<in T>(T x, T y);

* T >> O tipo dos objetos a serem comparados.Este parâmetro de tipo é contravariante. Ou seja, você pode usar o tipo que você especificou ou qualquer tipo que seja menos derivado.
* X >> O primeiro objeto a comparar.
* Y >> O segundo objeto para comparar.

private static int CompareUsuarios(Usuario e1, Usuario e2)

{

var comparar = e1.Nome.Length.CompareTo(e2.Nome.Length);

Console.WriteLine(e1.Nome + " CompareTo " + e2.Nome + ":" + comparar);

return comparar;

}

del\_comparison<Usuario> del\_03 = CompareUsuarios;

var romeu = new Usuario("Romeu", 10);

var julieta = new Usuario("Julieta", 90);

var nome = del\_03(romeu, julieta);

Console.WriteLine(nome);

Valor de retorno é um número inteiro(Int32) que indica os valores relativos de x e y , conforme mostrado na tabela a seguir.

|  |  |
| --- | --- |
| **Valor** | **Significado** |
| Menos que 0 | x é menor que y . |
| 0 0 | x é igual a y . |
| Maior que 0 | x é maior que y . |

public class Usuario

{

public string Nome { get; set; }

public int Idade { get; set; }

public Usuario(string nome, int idade)

{

this.Nome = nome;

this.Idade = idade;

}

}

class Program

{

public delegate int del\_comparison<in T>(T x, T y);

static void Main(string[] args)

{

del\_comparison<Usuario> del\_03 = CompareUsuarios;

var romeu = new Usuario("Romeu", 10);

var julieta = new Usuario("Julieta", 90);

var nome = del\_03(romeu, julieta);

Console.WriteLine(nome);

Console.WriteLine("======Comparison Nome Usuarios List======");

Comparison<Usuario> UsuariosComparer = new Comparison<Usuario>(CompareUsuarios);

List<Usuario> users = new List<Usuario>()

{

new Usuario("Romeu", 10),

new Usuario("Julieta", 90),

new Usuario("Hamlet", 55)

};

users.Sort(UsuariosComparer);

foreach (var p in users)

Console.WriteLine("Nome = " + p.Nome + " e Idade = " + p.Idade);

Console.WriteLine();

Console.WriteLine("======Comparison Anonima Idade Usuarios List======");

Comparison<Usuario> AnomIdadeComparer = new Comparison<Usuario>((Usuario u1, Usuario u2) =>

{

return u2.Idade.CompareTo(u1.Idade);

});

users.Sort(AnomIdadeComparer);

foreach (var p in users)

Console.WriteLine("Nome = " + p.Nome + " e Idade = " + p.Idade);

Console.WriteLine();

Console.WriteLine("======Comparison Lambda Lenght Usuarios Array======");

// Repare que é utilizado o objeto users com o ConvertAll

List<string> nomes = users.ConvertAll<string>(new Converter<Usuario, string>(ConvertUsuario\_ToString));

var array\_num = nomes.ToArray();

// Use lambda to sort on length.

Array.Sort(array\_num, (a, b) => a.Length.CompareTo(b.Length));

Console.WriteLine("RESULT: {0}", string.Join(";", array\_num));

Console.ReadKey();

}

public static string ConvertUsuario\_ToString(Usuario user)

{

return user.Nome;

}

private static int CompareUsuarios(Usuario e1, Usuario e2)

{

var comparar = e1.Nome.Length.CompareTo(e2.Nome.Length);

Console.WriteLine(e1.Nome + " CompareTo " + e2.Nome + ":" + comparar);

return comparar;

}

}

**Problemas com Delegate**

Os delegates têm alguns problemas que podem ser resolvidos com os eventos. Esses problemas são:

1. Qualquer um pode usar um operador de atribuição que pode sobrescrever as referências de métodos.

static void Main(string[] args)

{

//1. Qualquer um pode usar um operador de atribuição

// que pode sobrescrever as referências de métodos.

Console.WriteLine();

Console.WriteLine("======Problema delegate: Sobrescreve metodo associados======");

Action act = Display;

act += Show;

act += Display;

act();

Action act\_erro = Display;

act\_erro += Show;

Console.WriteLine("Aqui não tem o +");

act\_erro = Display;

act\_erro();

Console.ReadKey();

}

static void Display()

{

Console.WriteLine("Display");

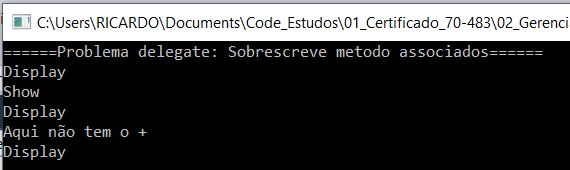
}

static void Show()

{

Console.WriteLine("Show");

}



1. O delegado pode ser chamado em qualquer lugar do código, o que pode violar a regra do encapsulamento.

private class Room

{

public Action<int> OnHeatAlert;

int temp;

public int Temperature

{

get { return this.temp; }

set

{

temp = value;

if (temp > 60)

{

if (OnHeatAlert != null)

OnHeatAlert(temp);

}

}

}

}

static void Main(string[] args)

{

//2. O delegado pode ser chamado em qualquer lugar do código,

// o que pode violar a regra do encapsulamento.

Console.WriteLine();

Console.WriteLine("======Problema delegate: Pode ser chamado em qualquer lugar do código======");

Room room = new Room();

room.OnHeatAlert = Alarm;

// OnHeatAlert será chamado

room.Temperature = 90;

room.Temperature = 15;

// OnHeatAlert será chamado novamente o que não deveria ocorrer

// porque o quarto não está quente (temp> 60) no set do valor Temperature

// Temperature é propriedade de Room. Delegado é chamado fora da classe Room

room.OnHeatAlert(room.Temperature);

Console.ReadKey();

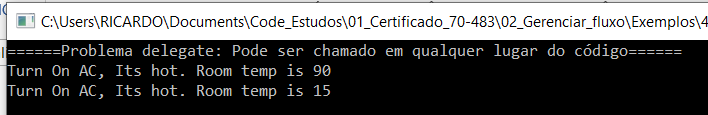
}

static void Alarm(int temp)

{

Console.WriteLine("Turn On AC, Its hot. Room temp is {0}", temp);

}



**Sumário**

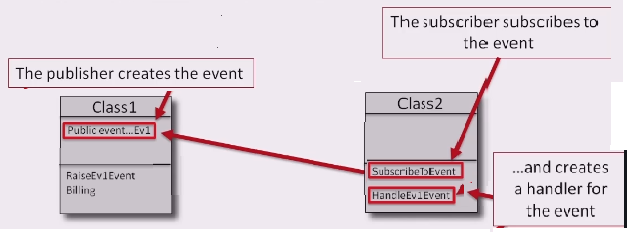
* Delegates são ponteiros de função. Eles armazenam a referência de método (s) dentro de um delegate.
* Delegate pode ser chamado em qualquer lugar do código para chamar o (s) método (s).
* A covariância no delegate é aplicada no tipo de retorno de um método.
* A contravariância no delegate é aplicada no parâmetro de entrada de um método.
* O Action delegate armazena a referência de um método que não retorna um valor.
* O Func delegate armazena a referência de um método que retorna um valor.
* O Predicate delegate armazena a referência de um método que usa um parâmetro de entrada e retorna um valor bool.
* A expressão Lambda é usada para criar um método anônimo.

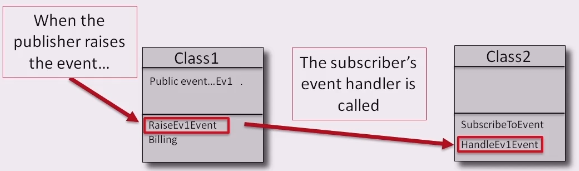
**EVENTS**

Eventos permitem que uma classe ou objeto notifique outras classes ou objetos quando ocorrer algo de interesse.Evento é uma ação que é executada quando uma condição especificada é satisfeita. Notifica todos os seus assinantes sobre a ação que será executada. Por exemplo, quando um evento do Windows 10 foi lançado, a Microsoft notificou todos os clientes para atualizar seu SO gratuitamente. Portanto, neste caso, a Microsoft é uma publicadora que lançou (levantou) um evento do Windows 10 e notificou os clientes sobre o assunto, e os clientes são os assinantes do evento e participaram do evento.

Da mesma forma, o evento C# é usado na classe para fornecer notificações aos clientes dessa classe quando algo acontece com seu objeto. Eventos são declarados usando delegates. Portanto, uma classe que contém a definição de um evento e seu representante é chamada Publisher. Por outro lado, uma classe que aceita o evento e fornece um manipulador de eventos é chamada Subscriber.

* A classe que envia (ou chamam) o evento é chamado de Publisher
* As classes que recebem (ou manipulam) os eventos são chamadas de Subscribers





Um padrão Publisher-Subscriber no desenvolvimento de aplicativos é bastante popular, pois é uma solução reutilizável para um problema recorrente. Você pode ser Subscriber de um evento e ser notificado quando o Publisher gerar um novo evento. Isso é usado para estabelecer um acoplamento loose entre os componentes de um aplicativo. Os eventos têm as seguintes características ...

* O publisher determina quando um evento é gerado
* Os subscribers determinam que ação é tomada em resposta ao evento
* Um evento pode ter vários subscribers
* Um subscriber pode manipular vários eventos de vários publishers
* Os eventos geralmente são usados ​​para sinalizar ações do usuário
* Os eventos são baseados no delegado EventHandler e na classe base EventArgs
* Evento sempre é um membro de dados de uma classe ou um struct. Não pode ser declarado dentro de um método.

Delegados formam a base para o sistema de eventos em C#. O exemplo abaixo mostra uma abordagem incorreta de uma classe expondo e delegate público invés de evento.

public class Publisher

{

public Action<string> OnChangeParam;

public void Raise(string str)

{

if (OnChangeParam != null)

{

OnChangeParam(str);

}

}

}

public class Subscriber

{

//Metodo ao ser adicionado ao evento/delegate

public void OnMethodName(string a)

{

Console.WriteLine("Event raised to method " + a);

}

}

public static void CreateAndRaise()

{

Publisher publisher = new Publisher();

Subscriber subscriber = new Subscriber();

publisher.OnChangeParam += subscriber.OnMethodName;

publisher.OnChangeParam += subscriber.OnMethodName;

// Sobrescreveu os outros métodos atribuídos

publisher.OnChangeParam = (str) => Console.WriteLine("Event raised to method 3" + str);

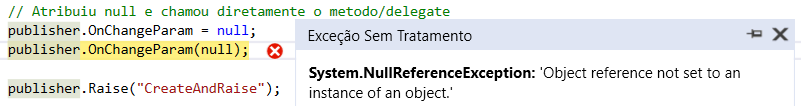
// Atribuiu null e chamou diretamente o metodo/delegate

publisher.OnChangeParam = null;

publisher.OnChangeParam(null);

publisher.Raise("CreateAndRaise");

}



Ao chamar CreateAndRaise, seu código cria uma nova instância do Publisher, assina o evento com dois métodos diferentes e, em seguida, gera o evento chamando publisher.Raise. A classe Publisher não tem conhecimento de nenhum assinante. Isso apenas levanta o evento. Se não houvesse assinantes para um evento, a propriedade OnChangeParam seria nula. É por isso que o método Raise verifica se o OnChangeParam não é nulo.

Embora esse sistema funcione, existem algumas fraquezas, pois nada impede que usuários externos da classe chamem diretamente publisher.OnChangeParam(null) para qualquer os assinantes. Além disso, permite também a inscrição do método usando = em vez de += ou -=, ocorrendo substituição dos outros métodos atribuídos anteriormente.

Vantagens dos Eventos em relação aos delegates:

1. Event encapsula um delegate; evita a substituição de uma referência de método restringindo o uso de atribuição do operador =.

* O evento sempre é inscrito usando += (Ex.: object.EventName + = OnMethodName);
* O evento é cancelado com o uso de -= (Ex.: object.EventName -= OnMethodName);
* Não pode object.EventName = OnMethodName

1. Ao contrário do delegate, o evento não pode ser chamado fora da classe, o que garante que o evento será chamado apenas quando uma determinada codificação estiver em conformidade.

É uma boa convenção de nomenclatura postfixar um nome de delegado personalizado com “EventHandler” somente quando for usado com o evento. Assim como prefixar o nome de métodos com On somente quando for usado com eventos, por exemplo, OnAlert. A exemplo abaixo mostra um exemplo modificado da classe Publisher que usa a sintaxe do evento.

public class Publisher\_Event

{

public event Action<string> ActionParEventHandler;

public event Func<string, bool> FuncEventHandler = delegate (string str) { return str == "2"; };

public void Raise(string str)

{

ActionParEventHandler(str);

var result = FuncEventHandler(str);

Console.WriteLine("result:" + result);

}

}

public class Subscriber

{

//Metodo ao ser adicionado ao evento/delegate

public void OnMethodName(string a)

{

Console.WriteLine("Event raised to method " + a);

}

}

public static void CreateAndRaise\_Event()

{

// Instancia um objeto publicador de evento e de assinante de evento

Publisher\_Event publisher = new Publisher\_Event();

Subscriber subscriber = new Subscriber();

publisher.ActionParEventHandler += subscriber.OnMethodName;

publisher.ActionParEventHandler += subscriber.OnMethodName;

//Não deixa sobrescrever os métodos

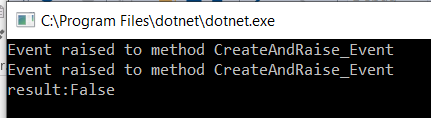
//publisher.ActionParEventHandler = subscriber.OnMethodName;

//Não deixa chamar diretamente o delegate E

//publisher.ActionParEventHandler("4");

publisher.Raise("CreateAndRaise\_Event");

}



Usando a sintaxe do evento, há algumas mudanças interessantes. Primeiro, você não está mais usando uma propriedade pública, mas um campo público. Normalmente, isso seria um passo para trás. No entanto, com a sintaxe do evento, o compilador protege seu campo de acesso indesejado.

Um evento não pode ser atribuído diretamente ao operador (com o = em vez de + =). Portanto, você não corre o risco de alguém remover todas as assinaturas anteriores, como na sintaxe do delegado.

Outra mudança é que nenhum usuário externo pode promover seu evento. Ele pode ser gerado apenas pelo código que faz parte da classe que definiu o evento.

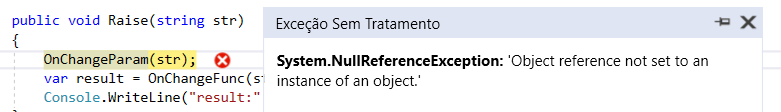
O C# fornece alguns delegates internos importantes para implementar eventos:

* **EventHandler**
* **PropertyChangedEventHandler**

Se não tivessevemos atribuído um valor ao declarar o delegado, assim:

public event Action<string> ActionParEventHandler;

Teríamos problema ao chamar o método de não tivéssemos atribuído nenhum assinante ao publicador:



Utilizando a sintaxe especial EventHandler ou EventHandler<T> para inicializar o evento em um delegate não atribuído valor, você pode remover a verificação nula em torno do aumento do evento, pois pode ter certeza de que o evento nunca é nulo. Usuários externos da sua classe não podem definir o evento como nulo; somente membros da sua classe podem. Desde que nenhum dos outros membros da classe defina o evento como nulo, você pode assumir com segurança que ele sempre terá um valor.

**EventHandler**

Como o manipulador de eventos agora usa dois parâmetros, é necessário revisar a declaração do evento, para que o delegado que ele usa reflete esses parâmetros. Você pode criar um novo delegado, mas o .NET Framework define um delegado genérico do EventHandler que facilita isso. Basta usar o tipo EventHandler e incluir o tipo de dados do segundo parâmetro, OverdrawnEventArgs neste exemplo, como o parâmetro de tipo do delegado genérico.

A Microsoft recomenda que todos os eventos forneçam dois parâmetros. O .NET Framework possui uma maneira padrão de manipular eventos que usa dois parâmetros: o EventHandler. O EventHandler é um evento definido no namespace System que é pré-conectado a um delegado que define um método do tipo de retorno nulo.

public delegate void EventHandler(object sender, EventArgs e);

Por padrão, é necessário um objeto sender e alguns argumentos de evento.

* **sender**>> primeiro parâmetro é de um tipo System.Object que se refere à instância (onde o evento foi definido) que gera o evento, ou nulo, se for proveniente de um método estático.
* **EventArgs**>> contém dados do evento. Se o evento não tiver dados para passar, o segundo parâmetro é simplesmente o valor do campo EventArgs.Empty. No entanto, se ele tiver um valor para passar, será encapsulado em um tipo derivado de EventArgs. Usando EventHandler <T>, você pode especificar o tipo de argumento do evento que deseja usar.

public class MyArgs : EventArgs

{

public MyArgs(int value)

{

Value = value;

}

public int Value { get; set; }

}

public class Publisher

{

public event EventHandler<MyArgs> OnChange = delegate { };

public void Raise()

{

OnChange(this, new MyArgs(42));

}

}

class Program

{

static void Main(string[] args)

{

CreateAndRaise();

Console.ReadKey();

}

public static void CreateAndRaise()

{

Publisher p = new Publisher();

p.OnChange += (sender, e) => Console.WriteLine("Event raised: {0}", e.Value);

p.Raise();

}

}

A classe Publisher usa um EventHandler <MyArgs>, que especifica o tipo dos argumentos do evento. Ao gerar esse evento, você deve passar uma instância do MyArgs. Os assinantes do evento podem acessar os argumentos e usá-los. Embora a implementação do evento use um campo público, você ainda pode personalizar a adição e remoção de assinantes. Isso é chamado de “custom event accessor”. O exemplo abaixo mostra um exemplo de criação de um acessador de evento personalizado para um evento.

public class Publisher\_Lock

{

private event EventHandler<MyArgs> \_onChange = delegate { };

public event EventHandler<MyArgs> ChangeEventHandler

{

add

{

lock (\_onChange)

{

\_onChange += value;

}

}

remove

{

lock (\_onChange)

{

\_onChange -= value;

}

}

}

public void Raise()

{

\_onChange(this, new MyArgs(42));

}

}

Um acessador de evento personalizado se parece muito com uma propriedade com um acessador get e set. Em vez de obter e definir, você usa adicionar e remover. É importante bloquear e adicionar e remover assinantes para garantir que a operação seja segura para threads.

Se você usar a sintaxe regular de eventos, o compilador gerará o acessador para você. Isso deixa claro que os eventos não são delegados; eles são um invólucro conveniente para os delegados. Os delegados são executados em uma ordem seqüencial. Geralmente, os delegados são executados na ordem em que foram adicionados, embora isso não seja especificado na CLI (Common Language Infrastructure), portanto, você não deve depender disso.

Uma coisa que é resultado direto da ordem seqüencial é como lidar com exceções. O exemplo abaixo mostra um exemplo em que um dos assinantes de eventos gera um erro.

public static void CreateAndRaiseError()

{

Publisher p = new Publisher();

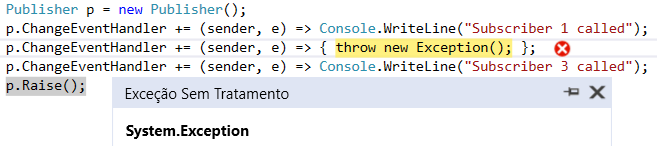
p.ChangeEventHandler += (sender, e) => Console.WriteLine("Subscriber 1 called");

p.ChangeEventHandler += (sender, e) => { throw new Exception(); };

p.ChangeEventHandler += (sender, e) => Console.WriteLine("Subscriber 3 called");

p.Raise();

}



Como você pode ver, o primeiro assinante é executado com êxito, o segundo gera uma exceção e o terceiro nunca é chamado. Se esse não é o comportamento que você deseja, é necessário aumentar manualmente os eventos e lidar com as exceções que ocorrerem. Você pode fazer isso usando o método GetInvocationList declarado na classe base System.Delegate. O exemplo abaixo mostra um exemplo de recuperação de assinantes e enumeração manual deles.

public class Publisher

{

public event EventHandler<MyArgs> ChangeEventHandler = delegate { };

public void Raise()

{

var exceptions = new List<Exception>();

foreach (Delegate handler in ChangeEventHandler.GetInvocationList())

{

try

{

handler.DynamicInvoke(this, new MyArgs(42));

}

catch (Exception ex)

{

exceptions.Add(ex);

}

}

if (exceptions.Any())

throw new AggregateException(exceptions);

}

}

public static void CreateAndRaiseError()

{

Publisher p = new Publisher();

p.ChangeEventHandler += (sender, e) => Console.WriteLine("Subscriber 1 called");

p.ChangeEventHandler += (sender, e) => { throw new Exception(); };

p.ChangeEventHandler += (sender, e) => Console.WriteLine("Subscriber 3 called");

try

{

p.Raise();

}

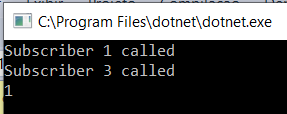
catch (AggregateException ex)

{

Console.WriteLine(ex.InnerExceptions.Count);

}

}



**PropertyChangedEventHandler**

PropertyChangedEventHandler é um delegate definido no namespace System.ComponentModel. É usado com evento para referir um método que será chamado sempre que uma Propriedade for alterada em um componente.

public delegate void PropertyChangedEventHandler(object sender, PropertyChangedEventArgs e)

O evento PropertyChanged usa um delegado PropertyChangedEventHandler na interface INotifyPropertyChanged. A classe, que implementa a interface INotifyPropertyChanged, deve definir a definição de evento PropertyChanged.

public class Person : INotifyPropertyChanged

{

private string name;

public event PropertyChangedEventHandler PropertyChanged;

public Person()

{

}

public string PersonName

{

get { return name; }

set

{

name = value;

// Call OnPropertyChanged whenever the property is updated

OnPropertyChanged("PersonName");

}

}

// Create the OnPropertyChanged method to raise the event

protected void OnPropertyChanged(string name)

{

PropertyChangedEventHandler handler = PropertyChanged;

if (handler != null)

{

handler(this, new PropertyChangedEventArgs(name));

}

}

}

class Program

{

static void Main(string[] args)

{

Person person = new Person();

person.PropertyChanged += OnPropertyChanged;

person.PersonName = "Ali";

Console.ReadKey();

}

private static void OnPropertyChanged(object sender, PropertyChangedEventArgs e)

{

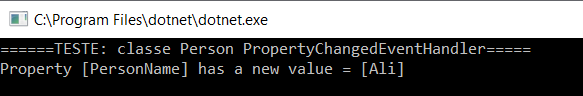
Person person = (Person)sender;

Console.WriteLine("Property [{0}] has a new value = [{1}]",

e.PropertyName, person.PersonName);

}

}



1. Implementar manipulação de exceções
   * Manipular tipos de exceções, incluindo exceções SQL(SQLException), exceções de rede (NetworkException/TransactionException) , exceções de comunicação(CommunicationException) , exceções de tempo limite de rede (TimeoutException); usar declarações de captura; usar uma classe base de uma exceção; implementar blocos try-catch-finally; lançar exceções; relançar uma exceção; criar exceções personalizadas; manipular exceções internas (InnerException); manipular exceções agregadas (AggregateException)

**MANIPULAÇÃO DE EXCEÇÃO**

Não importa quão bem você projete um aplicativo, os problemas ainda são inevitáveis. Os usuários inserem valores inválidos, os arquivos indispensáveis ​​serão excluídos, talvez você queira gravar um arquivo no disco e o disco esteja cheio, conexões críticas de rede falharão, você tenta se conectar a um banco de dados, mas o servidor de banco de dados não está disponível ou existe outra condição inesperada. Para evitar e se recuperar desses tipos de problemas, um programa deve executar:

* **Verificação de erros**
* **Tratamento de exceções.**

**VERIFICAÇÃO DE ERROS (veja mais detalhes no arquivo 4-Depurar >> Validando entradas)**

A verificação de erros é o processo de antecipar erros, verificar se eles ocorrerão e contorná-los. Por exemplo, se o usuário precisar inserir um número inteiro em uma caixa de texto, eventualmente alguém inserirá um valor não numérico. Se o programa tentar analisar o valor como se fosse um número inteiro, ele falhará. Em vez de travar, o programa deve validar o texto para ver se faz sentido antes de tentar analisá-lo. O método int.TryParse faz as duas coisas, tentando analisar um valor de texto e retornando um indicador de erro se ele falhar.

Se você puder, geralmente é melhor procurar proativamente problemas antes que eles ocorram, em vez de reagir a eles depois que eles acontecem. Você pode reduzir a necessidade desse tipo de verificação de erro removendo oportunidades para o usuário inserir valores inválidos. Por exemplo, se o programa usar um controle NumericUpDown ou TrackBar em vez de um TextBox, o usuário não poderá inserir um número inteiro inválido, como "dez" ou "1,2".

Da mesma forma, antes de abrir um arquivo ou fazer o download de um arquivo em uma rede, o programa pode verificar se o arquivo existe e se a conexão de rede está presente. Se o programa detectar esse tipo de erro, poderá informar ao usuário e cancelar qualquer operação que esteja tentando, em vez de apenas tentar abri-lo e manipular um erro se o arquivo não estiver lá.

**TRATAMENTO DE EXCEÇÕES**

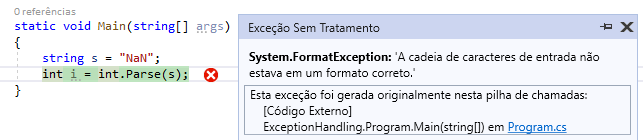
Ao contrário da verificação de erros, o tratamento de exceções é o processo de proteger o aplicativo quando ocorre um erro inesperado. Mesmo que você valide a entrada do usuário, procure os arquivos e conexões de rede necessários e verifique todos os outros erros que puder imaginar, o programa ainda poderá encontrar situações inesperadas. Um arquivo pode ficar corrompido; uma conexão de rede que estava presente pode falhar; o sistema pode ficar sem memória; ou uma biblioteca de códigos que você está usando e sobre a qual você não tem controle pode lançar uma exceção.

Em vez de trabalhar com códigos de erro, o .NET Framework usa exceções para sinalizar erros. Nesses casos, um programa pode se proteger usando blocos try-catch-finally. Você também pode usar essas exceções para sinalizar erros que ocorrem em seus próprios aplicativos e até criar tipos de exceção personalizados para sinalizar erros específicos.

É importante saber como trabalhar com exceções para que você possa implementar uma estratégia bem projetada para lidar ou gerar erros. A geração de informações de exceção também adiciona uma sobrecarga extra ao programa; portanto, você geralmente obtém melhor desempenho se antecipar erros antes que eles aconteçam.

Manipulando exceções

Quando ocorre um erro em algum lugar de um aplicativo, uma exceção é gerada. As exceções têm algumas vantagens em comparação com os códigos de erro. Uma exceção é um objeto em si que contém dados sobre o erro que ocorreu. Ele não apenas possui uma mensagem amigável, mas também contém o local em que o erro ocorreu e pode até armazenar dados extras, como um endereço em uma página que oferece alguma ajuda. Se uma exceção não for tratada, fará com que o processo atual seja encerrado. Como o exemplo abaixo onde o aplicativo gera um erro e é encerrado.



O método int.Parse lança uma exceção do tipo FormatException quando a sequência não é um número válido. Lançar uma exceção interrompe a execução do seu aplicativo. Em vez de continuar na linha a seguir, o tempo de execução percorre através da pilha de chamadas, procurando pelo código que captura e manipula a exceção. Se esse local não puder ser encontrado, a exceção não será tratada e encerrará o aplicativo.

Você não deve lançar exceções ao lidar com as situações esperadas. Você sabe que quando os usuários começam a inserir informações em seu aplicativo, eles cometem erros. Talvez eles digitam um número no formato errado ou se esqueçam de inserir um campo obrigatório. Não é recomendável criar uma exceção para esses tipos de situações esperadas.O tratamento de exceções altera o fluxo normal esperado do seu programa. Isso torna mais difícil ler e manter o código que usa exceções, especialmente quando elas são usadas em situações normais.

O uso de exceções também gera um leve impacto no desempenho. Como o tempo de execução precisa pesquisar todos os blocos externos de captura até encontrar um bloco correspondente e, quando não for, deve procurar se um depurador está conectado, leva um pouco mais de tempo para manipular. Quando ocorrer uma situação real inesperada que encerre o aplicativo, isso não será um problema. Mas para o fluxo regular do programa, isso deve ser evitado. Em vez disso, você deve ter a validação adequada e não confiar apenas em exceções.

Quando você precisa lançar uma exceção, é importante saber quais exceções já estão definidas no .NET Framework. Como os desenvolvedores estão familiarizados com essas exceções, eles devem ser usados ​​sempre que possível. Algumas exceções são lançadas apenas pelo tempo de execução. Você não deve usar essas exceções em seu próprio código. A tabela abaixo lista essas exceções.

|  |  |
| --- | --- |
| **Nome** | **Descrição** |
| ArithmeticException | Uma classe base para outras exceções que ocorrem durante operações aritméticas. |
| ArrayTypeMismatchException | Lançado quando você deseja armazenar um elemento incompatível dentro de uma matriz. |
| DivideByZeroException | Lançado quando você tenta dividir um valor por zero. |
| IndexOutOfRangeException | Lançado quando você tenta acessar uma matriz com um índice menor que zero ou maior que o tamanho da matriz. |
| InvalidCastException | Lançado quando você tenta converter um elemento em um tipo incompatível. |
| NullReferenceException | Lançado quando você tenta fazer referência a um elemento que é nulo. |
| OutOfMemoryException | Lançado ao criar um novo objeto falha porque o CLR não possui memória suficiente disponível. |
| OverflowException | Lançada quando uma operação aritmética transborda em um contexto verificado. |
| StackOverflowException | Lançado quando a pilha de execução está cheia. Isso pode acontecer em uma operação recursiva que não sai. |
| TypeInitializationException | Lançado quando um construtor estático lança uma exceção que não é tratada. |

Para lidar com uma exceção, você pode usar uma instrução try/catch/finally.

**Blocos try-catch-finally**

O bloco try-catch- finally permite que um programa capture erros inesperados e lide com eles. Na verdade, esse bloco consiste em três seções: uma seção try, uma ou mais seções de catch e uma seção finally.

* Deve ser usado em torno de declarações que possam causar exceção
* Exceções são representadas por classes derivadas de Exception
* Esta classe identifica o tipo de exceção e contém propriedades que possuem detalhes sobre a exceção

A seção try é necessária e você deve incluir o código que pode gerar uma exceção com uma instrução try. A quantidade de código que você coloca dentro de cada bloco try depende da situação. Você pode até aninhar outras seqüências try/catch/finally dentro de uma seção try para detectar erros sem sair da seção try original. Se você tiver várias instruções que podem lançar as mesmas exceções que precisam ser tratadas de maneira diferente, elas deverão estar em blocos try diferentes.

Você não precisa incluir catch ou finally e não precisa incluir nenhum código na seção catch ou finally. No C# 1, você também pode usar um bloco de captura sem um tipo de exceção. Isso pode ser usado para capturar exceções lançadas de outros idiomas, como C++, que não herdam de System.Exception (em C++, você pode lançar exceções de qualquer tipo). Atualmente, cada exceção que não herda de System.Exception é agrupada automaticamente em um System.Runtime.CompilerServices.RuntimeWrappedException. Como essa exceção herda de System.Exception, não é mais necessário o bloco catch vazio. É importante garantir que seu aplicativo esteja no estado correto quando o bloco catch terminar. Isso pode significar que você precisa reverter as alterações feitas pelo seu bloco try antes da exceção ser lançada.

string s = null;

try

{

int i = int.Parse(s);

}

catch

{

Console.WriteLine("The quantity must be an integer.");

}

Este código tenta analisar o valor da variável, se o valor não for um número inteiro, a instrução int.Parse emitirá uma exceção e a seção catch exibirá uma mensagem. Nesse caso, apenas uma mensagem é apropriada, independentemente da exceção lançada. Após a instrução try, você pode adicionar vários blocos de catch diferentes.

Você deve evitar usar diretamente a classe base Exception ao capturar e lançar exceções. Em vez disso, você deve tentar usar a exceção mais específica disponível. A tabela abaixo mostra exceções populares no .NET Framework que você pode usar em seus próprios aplicativos.

|  |  |
| --- | --- |
| **Nome** | **Descrição** |
| Exception | A classe base para todas as exceções. Tente evitar lançar e capturar essa exceção porque é muito genérica. |
| ArgumentException | Lance essa exceção quando um argumento para o seu método for inválido. |
| ArgumentNullException | Uma forma especializada de ArgumentException que você pode lançar quando um de seus argumentos for nulo e isso não for permitido. |
| ArgumentOutOfRangeException | Uma forma especializada de ArgumentException que você pode lançar quando um argumento estiver fora do intervalo permitido de valores. |
| FormatException | Lance essa exceção quando um argumento não tiver um formato válido. |
| InvalidOperationException | Lance essa exceção quando um método for chamado inválido para o estado atual do objeto. |
| NotImplementedException | Essa exceção é frequentemente usada no código gerado, em que um método ainda não foi implementado. |
| NotSupportedException | Lance essa exceção quando um método for invocado que você não oferece suporte. |
| ObjectDisposedException | Lance quando um usuário da sua classe tenta acessar métodos quando Dispose já foi chamado. |

O exemplo abaixo mostra um exemplo de captura de FormatException.

string s = "NaN";

try

{

int i = int.Parse(s);

}

catch (FormatException)

{

Console.WriteLine("{0} is not a valid number. Please try again", s);

}

Se você incluir o ExceptionType, a variável é uma variável da classe ExceptionType que fornece informações sobre a exceção. Todas as classes de exceção fornecem uma propriedade Message que fornece informações textuais sobre a exceção. Às vezes, você pode exibir essa mensagem para o usuário, mas geralmente a mensagem é técnica o suficiente para ser confusa para os usuários. A tabela abaixo lista as propriedades da classe base System.Exception.

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| StackTrace | Uma sequência que descreve todos os métodos atualmente em execução. Isso fornece uma maneira de rastrear qual método gerou a exceção e como esse método foi alcançado. |
| InnerException | Quando uma nova exceção é lançada porque ocorreu outra exceção, as duas são vinculadas à propriedade InnerException. |
| Message | Uma mensagem (esperançosamente) amigável para humanos que descreve a exceção. |
| HelpLink | Um nome de recurso uniforme (URN) ou localizador de recurso uniforme (URL) que aponta para um arquivo de ajuda. |
| HResult | Um valor de 32 bits que descreve a gravidade de um erro, a área na qual a exceção ocorreu e um número exclusivo para a exceção Esse valor é usado apenas ao cruzar limites gerenciados e nativos. |
| Source | O nome do aplicativo que causou o erro. Se a Origem não estiver definida explicitamente, o nome do assembly será usado. |
| TargetSite | Contém o nome do método que causou a exceção. Se esses dados não estiverem disponíveis, a propriedade será nula. |
| Data | Um dicionário de pares de chave/valor que você pode usar para armazenar dados extras para sua exceção. Esses dados podem ser lidos por outros blocos de captura e podem ser usados ​​para controlar o processamento da exceção. |

Ao usar um bloco catch, você pode usar um tipo de exceção e um identificador nomeado. Dessa forma, você efetivamente cria uma variável que manterá a exceção para você, para poder inspecionar suas propriedades. O exemplo abaixo mostra como fazer isso.

try

{

string s = Console.ReadLine();

int i = int.Parse(s);

Console.WriteLine("Parsed: {0}", i);

}

catch (FormatException e)

{

Console.WriteLine("Message: {0}",e.Message);

Console.WriteLine("StackTrace: {0}", e.StackTrace);

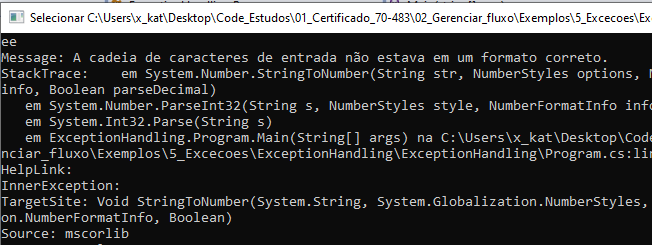
Console.WriteLine("HelpLink: {0}", e.HelpLink);

Console.WriteLine("InnerException: {0}", e.InnerException);

Console.WriteLine("TargetSite: {0}", e.TargetSite);

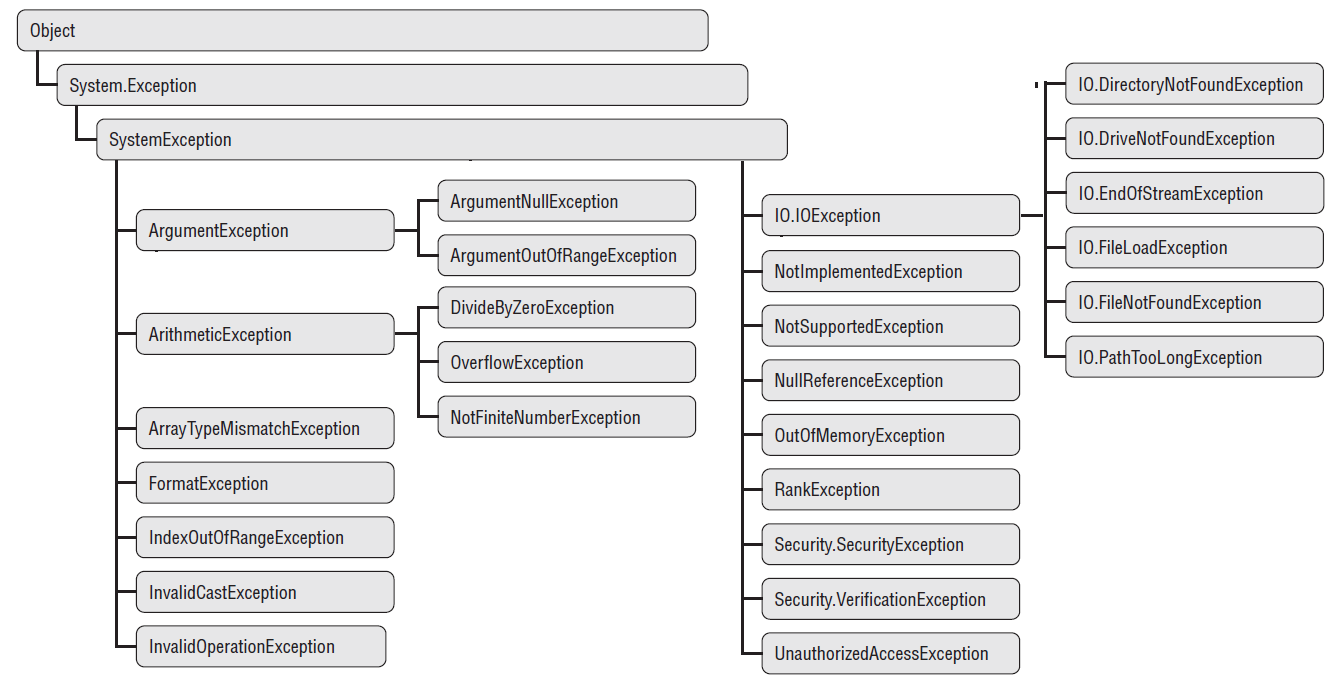
Console.WriteLine("Source: {0}", e.Source);

}



Se você incluir o ExceptionType, mas omitir a variável, a seção catch será executada para combinar os tipos de exceção mais específicos adicionando blocos catch extras. Como todas as exceções no .NET Framework são herdadas de System.Exception, você pode capturar todas as exceções possíveis capturando pelo tipo System.Exception. Os blocos catch devem ser especificados de mais específicos para menos específicos, porque esta é a ordem na qual o tempo de execução os examinará. Quando uma exceção é lançada, o primeiro bloco catch correspondente será executado. Se nenhum bloco de captura correspondente puder ser encontrado, a exceção ocorrerá.

O .NET Framework define centenas de classes de exceção para representar diferentes condições de erro. A figura abaixo mostra a hierarquia de algumas das classes de exceção mais comuns e úteis definidas no namespace System.



O código abaixo mostra um exemplo de captura de dois tipos de exceção diferentes.

string s = null;

try

{

int i = int.Parse(s);

}

catch (ArgumentNullException ae)

{

Console.WriteLine("You need to enter a value" + ae.Message);

}

catch (FormatException fe)

{

Console.WriteLine("{0} is not a valid number. {1}", s, fe.Message);

}

finally

{

Console.WriteLine("Program complete.");

Console.ReadLine();

}

Se a sequência s for nula, será lançada uma ArgumentNullException. Se a sequência não for um número, será lançada uma FormatException. Usando diferentes blocos de captura, quando encontra um ExceptionType correspondente, o programa executa as instruções dessa seção de captura, cada uma à sua maneira e ignora as seções de captura restantes.

Outro recurso importante do tratamento de exceções é a capacidade de especificar que determinado código sempre deve ser executado no caso de uma exceção. Isso pode ser feito usando o bloco finally junto com uma instrução try ou try/catch. A seção finally executa suas instruções quando as seções try e catch são concluídas, independentemente de como o código sai dessas seções. A seção finally sempre é executada, mesmo que o programa saia das seções try e catch por qualquer um dos seguintes motivos:

* O código na seção try é executado com êxito e nenhuma seção catch é executada.
* O código na seção try lança uma exceção e uma seção catch lida com isso.
* O código na seção try lança uma exceção e essa exceção não é capturada por nenhum seção catch.
* O código na seção try usa uma instrução de retorno para sair do método.
* O código em uma seção catch usa uma instrução de retorno para sair do método.
* O código em uma seção de captura gera uma exceção.

**Instrução Using**

A instrução using se comporta de fato como uma sequência de try-finally com finalidade especial que chama o método Dispose do objeto em sua seção finally. Por exemplo, considere o seguinte código:

internal class Pen : IDisposable

{

private Color color;

private int valor;

public Pen(Color cpr, int value)

{

this.color = cpr;

this.valor = value;

}

public void Dispose()

{

throw new NotImplementedException();

}

}

using (Pen pen = new Pen(Color.Red, 10))

{

// Use a caneta para desenhar ...

}

Isso é aproximadamente equivalente à seguinte sequência de try-finally:

Pen pen2 = new Pen(Color.Red, 10);

try

{

pen2 = new Pen(Color.Red, 10);

// Use a pen para desenhar ...

}

finally

{

if (pen2 != null) pen2.Dispose();

}

Isso significa que o programa chama o método Dispose da pen, não importa como ele saia do bloco using . Por exemplo, se as instruções no bloco executarem uma declaração de retorno ou lançarem uma exceção, o método Dispose ainda será chamado.

Obviamente, ainda existem situações em que um bloco finally não será executado. Por exemplo, quando o bloco try entra em um loop infinito, ele nunca sai do bloco try e nunca entra no bloco final. E em situações como falta de energia, nenhum outro código será executado. Todo o sistema operacional será encerrado. Há uma outra situação que você pode usar para impedir que um bloco finally seja executado. Obviamente, isso não é algo que você deseja usar regularmente, mas você pode ter uma situação em que apenas desligar o aplicativo é mais seguro do que executar blocos finally.

Impedir a execução do bloco finally pode ser alcançado usando o Environment.FailFast. Este método possui duas sobrecargas diferentes, uma que aceita apenas uma string e outra que aceita uma exceção. Quando esse método é chamado, a mensagem (e opcionalmente a exceção) é gravada no log de eventos do aplicativo Windows e o aplicativo é encerrado. A código abaixo mostra como você pode usar esse método.

string s = Console.ReadLine();

try

{

int i = int.Parse(s);

if (i == 42) Environment.FailFast("Special number entered");

}

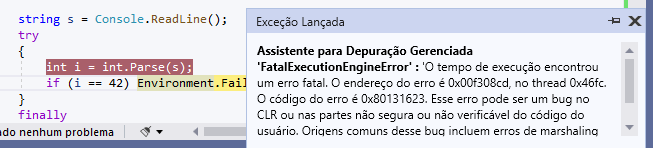
finally

{

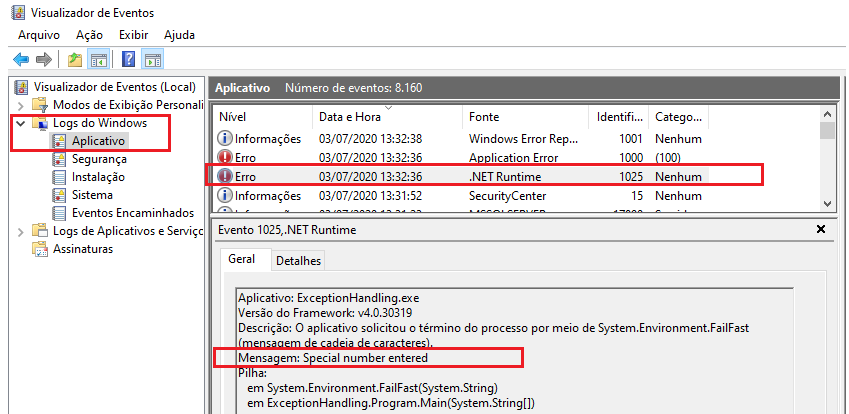
Console.WriteLine("Program complete finally.");

}

Se estiver em modo Debug emite o seguinte erro:



Mas quando você executa esse aplicativo sem um depurador conectado, uma mensagem é gravada no log de eventos. Para visualizar o log criado, abra o Visualizador de Eventos, indo em "Iniciar" no Windows >>“Administrative Tools”. Quando a janela estiver aberta, abra o aplicativo "Event Viewer".



A linha Programa concluído não será executada se 42 for inserido. Em vez disso, o aplicativo é encerrado imediatamente. É importante garantir que seu bloqueio finally não cause nenhuma exceção. Quando isso acontece, o controle sai imediatamente do bloco finally e passa para o próximo bloco externo, se houver. A exceção original foi perdida e você não pode mais acessá-la.

Você deve capturar apenas uma exceção quando puder resolver o problema ou quando desejar registrar o erro. Por esse motivo, é importante evitar bloqueios gerais nas camadas inferiores do seu aplicativo. Dessa forma, você pode perder acidentalmente uma exceção importante, mesmo sem saber que isso aconteceu. O registro também deve ser feito em algum lugar mais alto no seu aplicativo. Dessa forma, você pode evitar o registro de erros duplicados em várias camadas no seu aplicativo.

**Exceções SQL**

Além dessas exceções básicas, o .NET Framework define várias outras classes de exceção que têm uso mais especializado. Por exemplo, exceções SQL podem ocorrer quando um programa trabalha com bancos de dados do SQL Server. O SQL Server usa a classe única System.Data.SqlClient.SqlException para representar todos os erros e exceções. Você pode usar as propriedades do objeto SqlException para determinar o que deu errado e qual a gravidade. A tabela abaixo descreve algumas das propriedades da classe SqlException mais úteis.

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| Class | Um número entre 0 e 25, indicando o tipo de erro. |
| Class (0–10) | Mensagens informativas em vez de erros. e indicam problemas causados por erros nas informações inseridas por um usuário. |
| Class (11–16) | Problemas do usuário que podem ser corrigidos pelo usuário. |
| Class (17–19) | Você pode continuar trabalhando, embora talvez não seja possível executar uma determinada instrução.  17: O SQL Server ficou sem um recurso configurável, como bloqueios. O DBA pode  conserte isso.  18: Um problema de software interno não fatal.  19: O SQL Server excedeu um limite de recurso não configurável. |
| Class (20–25) | Os valores 20 a 25 são fatais e a conexão com o banco de dados é fechada.  20: Ocorreu um problema em uma declaração emitida pelo processo atual.  21: O SQL Server encontrou um problema que afeta todos os processos em um banco de dados.  22: Uma tabela ou índice foi danificado.  23: O banco de dados é suspeito.  24: Problema de hardware.  25: Erro do sistema. |
| LineNumber | Retorna o número da linha no lote de comandos T-SQL ou procedimento armazenado que causou o erro. |
| Message | Uma mensagem (esperançosamente) amigável para humanos que descreve a exceção. |
| Number | Retorna o número do erro. |
| Procedure | Retorna o nome do procedimento armazenado ou da chamada de procedimento remoto que causou o erro. |

Abaixo segue um exemplo de tratamento erro utilizando a classe SqlException:

string queryString = "EXECUTE NonExistantStoredProcedure";

StringBuilder errorMessages = new StringBuilder();

string connectionString = ConfigurationManager.ConnectionStrings["PeopleConnection"].ConnectionString;

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlCommand command = new SqlCommand(queryString, connection);

try

{

command.Connection.Open();

command.ExecuteNonQuery();

}

catch (SqlException ex)

{

for (int i = 0; i < ex.Errors.Count; i++)

{

errorMessages.Append("Index #" + i + "\n" +

"Message: " + ex.Errors[i].Message + "\n" +

"LineNumber: " + ex.Errors[i].LineNumber + "\n" +

"Source: " + ex.Errors[i].Source + "\n" +

"Procedure: " + ex.Errors[i].Procedure + "\n");

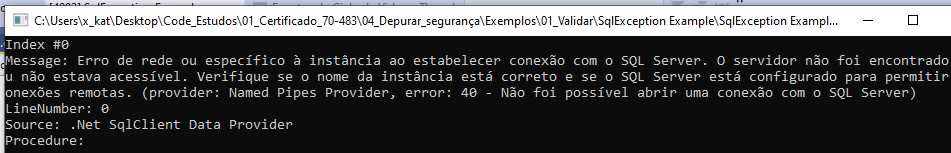
}

Console.WriteLine(errorMessages.ToString());

}

Console.ReadKey();

}



A classe System.Data.Common.DbException é a classe pai de SqlException e outras três classes que retornam informações semelhantes para outros tipos de banco de dados. A lista a seguir resume as outras três classes filho DbException filho:

* **System.Data.Odbc.OdbcException:** erros nos bancos de dados ODBC
* **System.Data.OleDb.OleDbException:** erros nos bancos de dados OLE DB
* **System.Data.OracleClient.OracleException:** erros nos bancos de dados Oracle

Todas essas classes fornecem uma propriedade Message que fornece informações sobre a exceção, embora não forneçam as propriedades Class, LineNumber, Number e Procedure fornecidas pela classe SqlException.

Além dos erros específicos do provedor, .NET Framework tipos de provedor de dados podem gerar .NET Framework exceções como System. OutOfMemoryexception e System. Threading. ThreadAbortException. A recuperação dessas exceções pode não ser possível.

Entrada inadequada pode fazer com que um .NET Framework tipo de provedor de dados gere uma exceção como System. ArgumentException ou System. IndexOutOfRangeException. Chamar um método na hora errada pode gerar System. InvalidOperationException.

Portanto, em geral, grave um manipulador de exceção que captura quaisquer exceções específicas de provedor, bem como exceções da Common Language Runtime. Eles podem ser dispostos em camadas da seguinte maneira:

try

{

// code here

}

catch (SqlException odbcEx)

{

// Handle more specific SqlException exception here.

}

catch (Exception ex)

{

// Handle generic ones here.

}

OU

try

{

// code here

}

catch (Exception ex)

{

if (ex is SqlException)

{

// Handle more specific SqlException exception here.

}

else

{

// Handle generic ones here.

}

}

**Exceções de Overflow**

Por padrão, um programa C# não lança uma exceção se uma operação aritmética causar um estouro de número inteiro. Se os operandos são integrais ou decimais, o programa descarta qualquer bit extra, retorna um resultado truncado e continua em execução como se nada tivesse dado errado. Nesse caso, você pode não estar ciente de que o resultado é sem sentido.

Você pode fazer o programa lançar uma OverflowException usando um bloco checked/unchecked ou usando a caixa de diálogo Configurações avançadas de compilação.

A palavra-chave checked adiciona exceções em estouros de número e erros podem ser evitados capturando o estouro mais cedo. Com unchecked, é o inverso, ele especifica que o estouro é um resultado aceitável de uma operação e nenhuma exceção é lançada. No exemplo abaixo, o tipo das variáveis é short, que não pode exceder o valor 32767. O programa incrementa o short e causa Overflow na variável estoura mas não em naoestoura, que retornaria um valor incorreto.

// The first short will overflow after the second short does.

short estoura = 0;

short naoestoura = 100;

try

{

//

// Keep incrementing the shorts until an exception is thrown.

// ... This is terrible program design.

//

while (true)

{

checked

{

estoura++;

}

unchecked

{

naoestoura++;

}

}

}

catch (OverflowException)

{

Console.WriteLine("OverflowException em estoura++.");

// Display the value of the shorts when overflow exception occurs.

Console.WriteLine(estoura); // 32767, valor mácimo de um short

Console.WriteLine(naoestoura); // -32669

}

De fato, o overflow de float e double nunca gera uma exceção, mas simplesmente retorna um valor especial de +/- Infinity. Por outro lado, para o tipo decimal, as palavras-chave checked/unchecked também são ignoradas, mas o excesso sempre gera uma OverflowException.

Quando uma operação de ponto flutuante causar um overflow(+) ou underflow(-) ou se produzir o valor especial NaN (que significa "não é um número") os tipos de ponto flutuante definem as propriedades estáticas PositiveInfinity, NegativeInfinity e NaN. Se você tentar colocar dentro de um número flutuante um número maior que float.MaxValue, será " IsInfinity". Você pode comparar uma variável de ponto flutuante com os valores PositiveInfinity e NegativeInfinity. Em vez de tentar comparar resultados com valores especiais, é melhor usar os métodos do tipo para determinar se uma variável possui um desses valores especiais. A tabela abaixo descreve esses métodos.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| IsInfinity | Retorna true se o valor for PositiveInfinity ou NegativeInfinity |
| IsNaN | Retorna true se o valor for NaN |
| IsNegativeInfinity | Retorna true se o valor for NegativeInfinity |
| IsPositiveInfinity | Retorna true se o valor for PositiveInfinity |

No entanto, se você comparar uma variável com NaN, o resultado será sempre falso. (Mesmo float.NaN == float.NaN retorna false). Um forma correta de fazer esta comparação seria como aseguir:

var zero = 0.0f;

// This will return true.

if (Single.IsNaN(0 / zero))

{

Console.WriteLine("Single.IsNan() can determine whether a value is not-a-number.");

}

// This will equal Infinity.

Console.WriteLine("Infinity plus 10.0 equals {0}.", (Single.PositiveInfinity + 10.0).ToString());

// This will return true.

Console.WriteLine("IsNegativeInfinity(-5.0F / 0) == {0}.", Single.IsNegativeInfinity(-5.0F / 0) ? "true" : "false");

O uso dos métodos de valor especial listados na tabela facilita a compreensão e a proteção do código, caso os valores especiais, como PositiveInfinity, sejam alterados em alguma versão posterior do .NET, por exemplo, se o tipo de dados flutuante passar para 64 bits.

**Lançando Exceções (Throwing Exceptions)**

Quando você deseja gerar um erro, primeiro precisa criar uma nova instância de uma exceção. Em C#, um objeto de exceção pode ser explicitamente lançado do código usando a palavra-chave throw. Um programador deve lançar uma exceção do código se uma ou mais das seguintes condições forem verdadeiras:

1. Quando o método não completa sua funcionalidade definida, por exemplo, Parâmetros possui valores nulos etc.
2. Quando uma operação inválida estiver em execução, por exemplo, tentando gravar em um arquivo somente leitura, etc.

Depois disso, o tempo de execução começará a procurar catch e, finalmente, os blocos. Se esse método interagir com o usuário, ele poderá exibir uma mensagem para informar ao usuário sobre o problema. O código abaixo mostra como você pode lançar uma exceção ArgumentNullException.

public static string OpenAndParse(string fileName)

{

if (string.IsNullOrWhiteSpace(fileName))

throw new ArgumentNullException("fileName", "Filename is required");

return File.ReadAllText(fileName);

}

string fileName = Console.ReadLine();

try

{

OpenAndParse(fileName);

}

catch (ArgumentNullException e)

{

Console.WriteLine($"Erro {e.Message}");

}

catch (Exception e)

{

Console.WriteLine("Message: {0}", e.Message);

Console.WriteLine("StackTrace: {0}", e.StackTrace);

Console.WriteLine("ExceptionType: {0}", e.GetType());

}

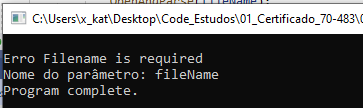
finally

{

Console.WriteLine("Program complete.");

Console.ReadLine();

}



No entanto, geralmente um método não deve interagir diretamente com o usuário. Por exemplo, se você estiver escrevendo uma biblioteca de ferramentas que serão chamadas por outros métodos, é provável que seus métodos não interajam diretamente com o usuário. Em vez disso, esses métodos devem lançar exceções próprias para informar o código de chamada que deu errado e, em seguida, deixar esse código lidar com o problema. Esse código pode exibir uma mensagem para o usuário ou pode resolver o problema sem incomodar o usuário. A se definir uma estratégia de excessão os seguintes quesitos devem ser consideradas:

* Quais exceções seu método deve capturar
* Quais exceções ele deve ignorar
* Quais exceções ele deve lançar

**Usando exceções e valores de retorno**

Um método pode executar alguma ação e, em seguida, retornar informações ao código de chamada através de um valor de retorno ou através de parâmetros de saída. As exceções fornecem um método para mais uma maneira de se comunicar com o código de chamada. Uma exceção informa ao programa que algo de excepcional aconteceu e que o método pode não ter concluído a tarefa que estava executando.

Há alguma discussão na Internet sobre quando um método deve retornar informações através de um valor ou parâmetros de retorno e quando deve retornar informações através de uma exceção. A maioria dos desenvolvedores concorda que as informações normais de status devem ser retornadas através de um valor de retorno e que exceções devem ser usadas apenas quando houver um erro.

A melhor maneira de decidir se deseja usar uma exceção é perguntar se o código de chamada deve ter permissão para ignorar o status do método. Se um método retornar informações de status através de seu valor de retorno, o código de chamada poderá ignorá-las. Se o método lançar uma exceção, o código de chamada deverá incluir um bloco trycatch para manipular a exceção explicitamente. Por exemplo, considere o seguinte método que retorna o fatorial de um número:

private long Factorial(long n)

{

// Make sure n >= 0.

if (n < 0) return 0;

checked

{

try

{

long result = 1;

for (long i = 2; i <= n; i++) result \*= i;

return result;

}

catch

{

return 0;

}

}

}

Há dois problemas com esta abordagem. Primeiro, o código de chamada pode ignorar o erro e tratar o valor 0 como fatorial de um número, fornecendo um resultado incorreto. Se o valor for usado em um cálculo complexo, o erro poderá ser incorporado no cálculo. O programa produziria um resultado incorreto que pode ser difícil de reparar e corrigir posteriormente.

O segundo problema é que o código de chamada não pode dizer o que deu errado. O valor de retorno 0 não indica se o parâmetro de entrada foi menor que 0 ou se houve um estouro de número inteiro. Você pode usar vários valores de retorno; portanto, 0 significa que o parâmetro era menor que 0 e –1 significa um excesso de número inteiro, mas isso apenas cria mais valores de status que o código de chamada pode ignorar.

Uma solução melhor é lançar exceções apropriadas quando apropriado. A seguinte versão do método fatorial, mostrada anteriormente neste capítulo, usa exceções:

private long Factorial\_Checked(long n)

{

// Make sure n >= 0.

if (n < 0) throw new ArgumentOutOfRangeException(

"n", "The number n must be at least 0 to calculate n!");

checked

{

long result = 1;

for (long i = 2; i <= n; i++) result \*= i;

return result;

}

}

Se o parâmetro for menor que zero, o código emitirá uma exceção. Como os cálculos são colocados em um bloco checked, se causarem um integer overflow, eles lançarão uma OverflowException.

Você não deve tentar reutilizar objetos de exceção. Cada vez que você lança uma exceção, deve criar uma nova, especialmente ao trabalhar em um ambiente multithread, o rastreamento de pilha da sua exceção pode ser alterado por outro thread. Ao capturar uma exceção, você pode optar por repetir a exceção. Você tem três maneiras de fazer isso:

1. **Use throw sem um identificador**
2. **Use throw com a exceção original**
3. **Use throw com uma nova exceção**
4. **Método ExceptionDispatchInfo.Throw**

**Use throw sem um identificador**

Reproduz novamente a exceção sem modificar a pilha de chamadas. Essa opção deve ser usada quando você não deseja modificações na exceção. O código abaixo mostra um exemplo de uso desse mecanismo.

[Conditional("DEBUG")]

private static void Log(Exception logEx)

{

Debug.WriteLine("Debug Log Message: {0}", logEx.Message);

Debug.WriteLine("Debug Log StackTrace: {0}", logEx.StackTrace);

Debug.WriteLine("ExceptionType: {0}", e.GetType());

}

public static void Rethrowing\_JustThrow(string fileName)

{

try

{

OpenAndParse(fileName);

}

catch (Exception logEx)

{

Log(logEx);

throw; // rethrow the original exception

}

}

public static void Main(string[] args)

{

string fileName = Console.ReadLine();

try

{

Rethrowing\_JustThrow(fileName);

}

catch (Exception e)

{

Console.WriteLine("Message: {0}", e.Message);

Console.WriteLine("StackTrace: {0}", e.StackTrace);

Console.WriteLine("ExceptionType: {0}", e.GetType());

}

finally

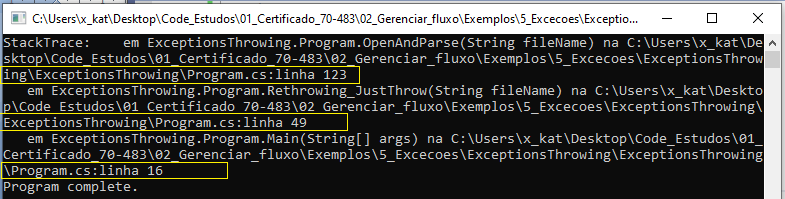
{

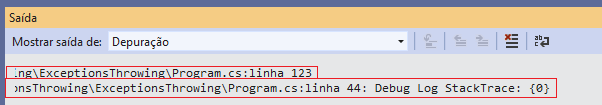
Console.WriteLine("Program complete.");

Console.ReadLine();

}

}





**Use throw com a exceção original**

Esta opção redefine a pilha de chamadas para o local atual no código. Portanto, você não pode ver de onde veio a exceção e é mais difícil depurar o erro. A nova versão lança explicitamente o mesmo objeto de exceção que o bloco try-catch apanhado. Quando o código lança uma exceção dessa maneira, a pilha de chamadas da exceção é redefinida para o local atual, para que se refira à linha de código que contém a instrução throw. Isso pode enganar todos os programadores que tentam localizar um problema, fazendo-os olhar para a linha de código errada. A situação é ainda pior se a linha de código que lançou a exceção estiver dentro de outro método chamado por este. Se você repetir a exceção dessa maneira, o fato de o erro estar em outro método será perdido.

private static void Rethrowing\_OriginalException(string fileName)

{

try

{

OpenAndParse(fileName);

}

catch (Exception logEx)

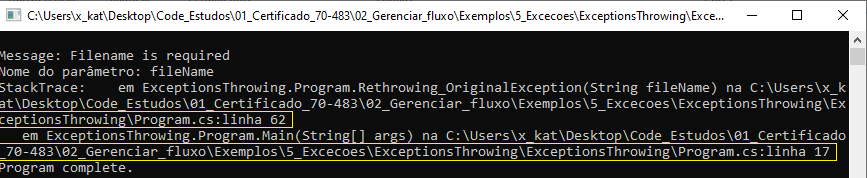
{

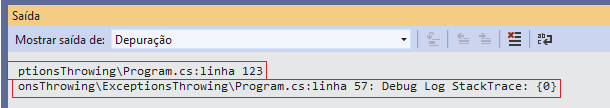
Log(logEx);

throw logEx;

}

}

****

****

**Use throw com uma nova exceção**

Uma outra maneira de preservar as informações de rastreamento da pilha da exceção original na repetição da reprodução é agrupar a exceção original com outra exceção. Esta opção pode ser útil quando você deseja gerar outra exceção para o chamador do seu código, como esta:

private static void Rethrowing\_NewException(string fileName)

{

try

{

OpenAndParse(fileName);

}

catch (Exception logEx)

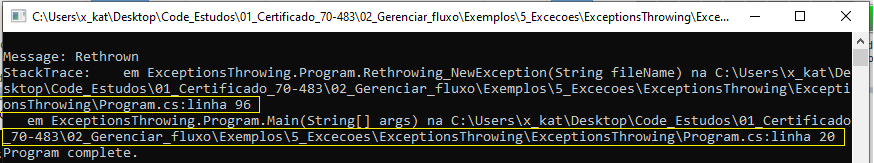
{

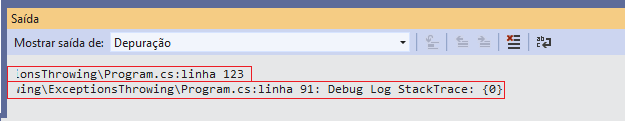
Log(logEx);

throw new Exception("Rethrown", logEx);

}

}





Diga, por exemplo, que você está trabalhando em um aplicativo de pedidos. Quando um usuário faz um pedido, você o coloca imediatamente em uma fila de mensagens para que outro aplicativo possa processá-lo. Quando ocorre um erro interno na fila de mensagens, uma exceção do tipo MessageQueueException é gerada. Para os usuários do seu aplicativo de pedidos, essa exceção não faz sentido. Eles não conhecem o funcionamento interno do seu módulo e não entendem de onde vem o erro na fila de mensagens.

Em vez disso, você pode lançar outra exceção, algo como uma OrderProcessingException personalizada e definir a InnerException como a exceção original. Em OrderProcessingException, você pode colocar informações extras para o usuário do seu código colocar o erro no contexto e ajudá-lo a resolvê-lo. O código abaico mostra um exemplo. A exceção original é preservada, incluindo o rastreamento de pilha, e uma nova exceção com informações extras é adicionada.

[Serializable]

public class OrderProcessingException : Exception, ISerializable

{

public int OrderId { get; }

public OrderProcessingException(string message, Exception innerException, int orderId)

: base(message, innerException)

{

OrderId = orderId;

this.HelpLink = "http://www.mydomain.com/infoaboutexception\u201d";

}

}

private static void Rethrowing\_AnotherNewException(string fileName)

{

try

{

OpenAndParse(fileName);

}

catch (MessageQueueException ex)

{

throw new OrderProcessingException("Error while processing order", ex, ex.ErrorCode);

}

}

Certifique-se de não perca nenhum detalhe de exceção ao repetir uma exceção. Lance uma nova exceção que aponte para a original quando você desejar adicionar informações extras; caso contrário, use a palavra-chave throw sem um identificador para preservar os detalhes da exceção original.

**Método ExceptionDispatchInfo.Throw**

No C# 5, uma nova opção é adicionada para relançar uma exceção. Você pode usar o método ExceptionDispatchInfo.Throw, que pode ser encontrado no espaço para nome System.Runtime.ExceptionServices. Este método pode ser usado para lançar uma exceção e preservar o rastreamento de pilha original. Você pode usar esse método mesmo fora de um bloco catch, como mostra a abaixo.

private static void Rethrowing\_CaptureThrow(string fileName)

{

try

{

OpenAndParse(fileName);

}

catch (Exception ex)

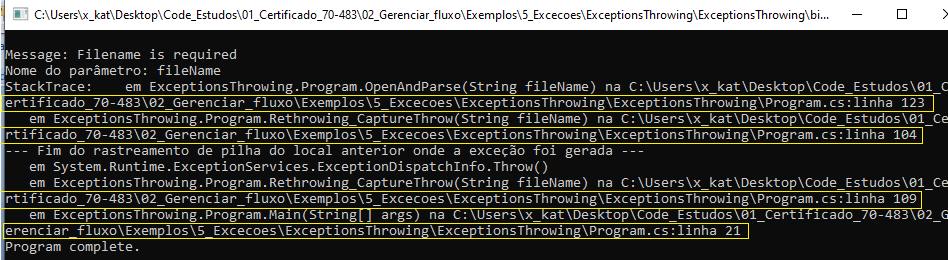
{

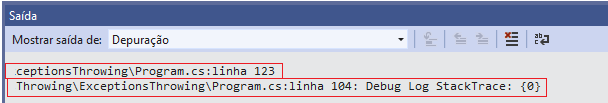
Log(ex);

ExceptionDispatchInfo.Capture(ex).Throw();

}

}





Ao examinar o rastreamento de pilha, você vê esta linha, que mostra onde o rastreamento de pilha de exceção original termina e o ExceptionDispatchInfo.Throw é usado:

**--- Fim do rastreio de pilha do local anterior onde a exceção foi gerada ---**

Esse recurso pode ser usado quando você deseja capturar uma exceção em um segmento e lançá-lo em outro segmento. Usando a classe ExceptionDispatchInfo, você pode mover os dados de exceção entre os threads e lançá-los. O .NET Framework usa isso ao lidar com o recurso assíncrono / aguardado adicionado no C# 5. Uma exceção lançada em um encadeamento assíncrono será capturada e retrocedida no encadeamento em execução.

Em geral, no .Net é desaconselhado fortemente o uso de throw ex para refazer a exceção em um bloco catch, pois destrói as informações sobre onde a exceção foi lançada originalmente e definitivamente causará frustração ao olhar para os logs e tentar descobrir fora o que tinha acontecido. Com o .NET Framework 4.5 e superior, eu sempre usava ExceptionDispatchInfo para retroceder, pois fornece a imagem mais completa dos eventos que aconteceram. Nas versões da estrutura inferiores a 4,5, eu usaria o throw como a maneira mais simples de rever novamente e manter as informações sobre a origem da exceção. O agrupamento da exceção por outra exceção para manter as informações sobre o método que levou à exceção simplesmente não vale a pena.

**Criando exceções personalizadas**

Depois que a exceção for lançada, se possível, você deve lançar uma das classes de exceção definidas pelo .NET Framework. As classes de exceção predefinidas têm significados específicos; portanto, se você usar uma, outros desenvolvedores terão uma boa idéia do que a exceção representa. Mas há situações em que você deseja usar uma exceção personalizada. Isso é especialmente útil quando os desenvolvedores que trabalham com seu código estão cientes dessas exceções e podem tratá-las de uma maneira mais específica do que as exceções da estrutura.

Uma exceção personalizada deve herdar de System.Exception. Você precisa fornecer pelo menos um construtor sem parâmetros. Também é uma prática recomendada adicionar outros construtores: um que aceita uma string, outro que aceita uma string e uma exceção e outro para serialização. O código a seguir mostra uma classe InvalidException simples que fornece quatro construtores que usam parâmetros semelhantes aos usados pelos construtores definidos na classe Exception:

[Serializable]

class InvalidProjectionException : Exception

{

public int OrderId { get; }

public InvalidProjectionException()

: base() { }

public InvalidProjectionException(string message)

: base(message)

{

this.HelpLink = "http://www.mydomain.com/infoaboutexception\u201d";

}

public InvalidProjectionException(string message, Exception innerException)

: base(message, innerException)

{

this.HelpLink = "http://www.mydomain.com/infoaboutexception\u201d";

}

protected InvalidProjectionException(SerializationInfo info, StreamingContext context)

: base(info, context)

{

OrderId = (int)info.GetValue("OrderId", typeof(int));

this.HelpLink = "http://www.mydomain.com/infoaboutexception\u201d";

}

public void GetObjectData(SerializationInfo info, StreamingContext context)

{

info.AddValue("OrderId", OrderId, typeof(int));

}

}

static void Main(string[] args)

{

try

{

Show();

}

catch (InvalidProjectionException ex)

{

Console.WriteLine(ex.Message);

}

Console.ReadKey();

}

private static void Show()

{

throw new InvalidProjectionException("It's a custom exception!");

}

Por convenção, você deve usar o sufixo Exception ao nomear todas as suas exceções personalizadas. Também é importante adicionar o atributo Serializable, que garante que sua exceção possa ser serializada e desserializada para cruzar os limites do AppDomain e funcione corretamente nos domínios de aplicativos (por exemplo, quando um serviço da Web retorna uma exceção). Cada um dos construtores simplesmente passa seus parâmetros para os construtores da classe base. Os tipos SerializationInfo e StreamingContext são definidos no namespace System.Runtime.Serialization.

Ao criar sua exceção personalizada, você pode decidir quais dados extras você deseja armazenar. A exposição desses dados por meio de propriedades pode ajudar os usuários de sua exceção a inspecionar o que deu errado. Se a classe de exceção personalizada fornecer informações especializadas, adicione as propriedades apropriadas à classe e adicione construtores adicionais, se desejar.

A Microsoft costumava recomendar que você derivasse novas classes de exceção do ApplicationException, mas depois decidiu que isso adicionaria outro nível à hierarquia de exceções sem fornecer nenhum benefício real. Você nunca deve herdar de System.ApplicationException. A idéia original era que todas as exceções de tempo de execução do C# fossem herdadas de System.Exception e todas as exceções personalizadas de System.ApplicationException. No entanto, como o .NET Framework não segue esse padrão, a classe se tornou inútil e perdeu seu significado. Não importa se você deriva novas classes de exceção de Exception ou ApplicationException, mas provavelmente vale a pena usar Exception para ser consistente com outros desenvolvedores que seguem as recomendações da Microsoft.

**Classe NetworkException/TransactionException**

??????????????????

**Classe CommunicationException**

Em clientes WCF, as falhas de SOAP que ocorrem durante a comunicação que são de interesse para aplicativos cliente são geradas como exceções gerenciadas. As exceções que podem ocorrer durante a execução de aplicativos que usam o modelo de programação de cliente do Windows Communication Foundation (WCF) são:

* **Exceções Inesperadas**: normalmente, não há uma maneira útil de lidar com erros inesperados, portanto, normalmente você não deve capturá-los ao chamar um método de comunicação do cliente WCF. Incluem falhas catastróficas como OutOfMemoryException e erros de programação como ArgumentNullException ou InvalidOperationException .
* **Exceções Esperadas:** indicam um problema durante a comunicação que pode ser manipulada com segurança anulando o cliente WCF e relatando uma falha de comunicação. Como fatores externos podem causar esses erros em qualquer aplicativo, os aplicativos corretos devem capturar essas exceções e recuperar quando ocorrerem. Os aplicativos WCF incluem exceções dos dois tipos a seguir como resultado da comunicação.

1. **TimeoutException:** são emitidos quando uma operação excede o período de tempo limite especificado.
2. **CommunicationException:** são lançados quando há alguma condição de erro de comunicação recuperável no serviço ou no cliente. A CommunicationException classe tem dois tipos derivados importantes:

* **FaultException:** são geradas quando um ouvinte recebe uma falha de SOAP que não é esperada ou especificada no contrato de operação; Geralmente isso ocorre quando o aplicativo está sendo depurado e o serviço tem a ServiceDebugBehavior.IncludeExceptionDetailInFaults propriedade definida como true .
* **FaultException<TDetail> :** tipo genérico, são geradas no cliente quando uma falha de SOAP especificada no contrato de operação é recebida em resposta a uma operação bidirecional (ou seja, um método com atributo OperationContractAttribute com IsOneWay definido como false ).

Como FaultException<TDetail> deriva de, FaultException e FaultException deriva de CommunicationException , é importante capturar essas exceções na ordem correta. Se, por exemplo, você tiver um bloco try/catch no qual você captura primeiro CommunicationException , todas as falhas de SOAP especificadas e não especificadas serão tratadas lá; quaisquer blocos catch subsequentes para manipular uma FaultException<TDetail> exceção personalizada nunca serão invocados. Portanto, para impedir que o manipulador de CommunicationException genérico detecte esses tipos de exceção mais específicos, Capture essas exceções antes de manipular CommunicationException.

?????????????????? Exemplo

**Classe TimeoutException**

A classe TimeoutException pode especificar uma mensagem para descrever a origem da exceção. Quando um método gera essa exceção, a mensagem é geralmente "o tempo limite fornecido expirou e a operação não foi concluída". A exceção é gerada quando o tempo alocado para um processo ou uma operação tiver expirado.

Essa classe é usada, por exemplo, pelo membro WaitForStatus da classe ServiceController. A operação que pode gerar a exceção é uma alteração da propriedade de Status do serviço (por exemplo, de Paused para ContinuePending).

O exemplo de código a seguir demonstra o uso de TimeoutException em conjunto com membros da classe System.IO.Ports.SerialPort.

string input;

try

{

// Set the COM1 serial port to speed = 4800 baud, parity = odd,

// data bits = 8, stop bits = 1.

SerialPort sp = new SerialPort("COM1", 4800, Parity.Odd, 8, StopBits.One);

// Timeout after 2 seconds.

sp.ReadTimeout = 2000;

sp.Open();

// Read until either the default newline termination string

// is detected or the read operation times out.

input = sp.ReadLine();

sp.Close();

// Echo the input.

Console.WriteLine(input);

}

catch (TimeoutException e)

{

Console.WriteLine(e);

}

**Classe AggregateException**

Exceções no .NET são o mecanismo fundamental pelo qual erros e outras condições excepcionais são comunicados. Com base no modelo de manipulação de exceção estruturada (SEH) do Windows, apenas uma exceção do .NET pode estar "em andamento" a qualquer momento em qualquer thread específico, afinal, uma operação normalmente gera apenas uma exceção e, portanto, no código seqüencial que escrevemos na maioria das vezes, precisamos nos preocupar com apenas uma exceção por vez.

AggregateException é usado para consolidar várias falhas em um único objeto de exceção rethrowável. Ele é usado extensivamente, mas não se limita a, na TPL (biblioteca paralela de tarefas) e no Parallel LINQ (PLINQ) qunado ocorrem alguma exceção. O .NET Framework lida com isso agregando todas as exceções em um AggregateException. Esta exceção expõe uma lista de todas as exceções que ocorreram durante a execução paralela. O exemplo abaixo mostra como você pode lidar com isso.

public static bool IsEven(int i)

{

if (i % 10 == 0) throw new ArgumentException("i");

return i % 2 == 0;

}

var numbers = Enumerable.Range(0, 20);

try

{

var parallelResult = numbers.AsParallel().Where(i => IsEven(i));

parallelResult.ForAll(e => Console.WriteLine(e));

}

catch (AggregateException e)

{

Console.WriteLine("There where {0} exceptions", e.InnerExceptions.Count);

}

Quando estamos fazendo o tratamento de exceção em um determinado programa, pode ser interressante primeiro classifica-los em quatro tipos:

* **Rxceções fatais:** não são sua culpa, você não pode evitá-las e não pode limpar sensivelmente delas. Elas quase sempre acontecem porque o processo está profundamente doente e está prestes a ser eliminado de sua miséria. Falta de memória, encadeamento interrompido e assim por diante. Não há absolutamente nenhum sentido em capturá-los, porque nada que seu código de usuário insignificante possa fazer resolverá o problema. Apenas deixe seus blocos "finalmente" correrem e espere o melhor.
* **Exceções desordenadas:** são sua própria falha, você poderia tê-las evitado e, portanto, são erros no seu código. Você não deve pegá-los; isso é ocultar um bug no seu código. Em vez disso, você deve escrever seu código para que a exceção não possa ocorrer em primeiro lugar e, portanto, não precise ser detectada. Esse argumento é nulo, o tipo de conversão é ruim, o índice está fora do intervalo, você está tentando dividir por zero - todos esses problemas que você poderia ter evitado com muita facilidade em primeiro lugar, portanto, evite a bagunça em primeiro lugar ao invés de tentar limpá-lo.
* **Exceções irritantes**: são o resultado de decisões infelizes de design. As exceções irritantes são lançadas em uma circunstância completamente não excepcional e, portanto, devem ser capturadas e manipuladas o tempo todo. O exemplo clássico de uma exceção irritante é Int32.Parse, que lança se você der uma string que não possa ser analisada como um número inteiro. Mas o caso de uso de 99% para esse método está transformando as seqüências de caracteres inseridas pelo usuário, o que pode ser algo antigo e, portanto, não é de forma alguma excepcional que a análise falhe. Pior, não há como o chamador determinar antecipadamente se o argumento é ruim sem implementar o método inteiro, caso em que não precisaria chamá-lo em primeiro lugar. Essa infeliz decisão de design foi tão irritante é claro que a equipe de estruturas implementou o TryParse logo em seguida, o que faz a coisa certa. Você precisa capturar exceções irritantes, mas fazê-lo é irritante. Tente nunca escrever uma biblioteca que gere uma exceção irritante.
* **Exceções exógenas:** parecem ser um pouco como exceções irritantes, exceto que elas não são o resultado de escolhas infelizes de design. Em vez disso, são o resultado de realidades externas desarrumadas que afetam sua lógica bonita e nítida do programa. Considere este código pseudo-C #, por exemplo:

var filename = @"C:\DummyFile.txt";

try

{

using (FileStream fs = File.Open(filename, FileMode.OpenOrCreate))

{

// Blah blah blah

}

}

catch (FileNotFoundException)

{

// Handle filename not found

}

Se eliminássemos o estrutura try-catch poderíamos ter uma situação de "condição de corrida". Algum outro processo poderia ter excluído, bloqueado, movido ou alterado as permissões do arquivo entre o FileExists e o OpenFile.

if (!File.Exists(filename)) ;

// Handle filename not found

else

{

using (FileStream fs = File.Open(filename, FileMode.OpenOrCreate))

{

// Blah blah blah

}

}

Podemos ser mais sofisticados? E se bloquearmos o arquivo com um lock? Isso não ajuda. A mídia pode ter sido removida da unidade, a rede pode ter caído. Você precisa capturar uma exceção exógena, porque sempre pode acontecer, por mais que você tente evitá-la; é uma condição exógena fora do seu controle. Entao, para resumir:

* Não pegue exceções fatais; nada que você possa fazer sobre eles, e tentar geralmente piora as coisas.
* Corrija seu código para que ele nunca desencadeie uma exceção irracional - uma exceção "índice fora do intervalo" nunca deve ocorrer no código de produção.
* Evite irritar exceções sempre que possível chamando as versões "Try" desses métodos irritantes que geram circunstâncias não excepcionais. Se você não puder evitar chamar um método vexatório, pegue suas exceções vexatórias.
* Sempre lide com exceções que indicam condições exógenas inesperadas; geralmente não vale a pena nem é prático antecipar todas as falhas possíveis. Apenas tente a operação e esteja preparado para lidar com a exceção.

**Sumário**

* Exceção é um erro que ocorre no tempo de execução e pode interromper a execução de um aplicativo.
* Os blocos try-catch-finalmente são úteis para lidar com as exceções normalmente.
* Programaticamente, uma exceção pode ser lançada usando uma palavra-chave throw.
* Um personalizado pode ser criado herdando a classe Exception.
* Expressão regular é útil para validar os valores grandes de cadeias com certos padrões.