Исключения и обработка ошибок

Юрий Литвинов
yurii.litvinov@gmail.com

06.03.2018г

1. Исключения, бросание исключений

Часть ошибок ловится на этапе компиляции, однако с остальными так или иначе приходится иметь дело во время выполнения. Каждый язык имеет свои механизмы и средства для обнаружения и обработки ошибочных ситуаций. В C, например, это было соглашение о возвращаемых функциями значениях + переменная errno, по которой можно было узнать о том, что же конкретно произошло. Недостатки данного подхода:

- это не особенность языка или среды, а лишь соглашение между разработчиками, на которое при желании вполне можно забить;
- можно забыть или не знать о каких-то особенностях возвращаемых значений;
- если проверять значение, возвращаемое каждым вызовом, читаемость кода резко падает.

Создавать большие, надежные и в то же время простые системы с таким подходом очень сложно.

Концепция обработки исключений начала зарождаться еще в 60-х годах с развитием операционных систем или даже с появлением в бейсике операции goto. Исключения в C# работают идеологически так же, как и исключения в Java, которые основаны на исключениях C++, которые в свою очередь берут начало от исключений языка Ada.

В том месте, где случилось что-то нехорошее, мы можем и не знать, что же с этой ситуацией делать. Но продолжать работу дальше нельзя — кто-то где-то (если не в этом контексте, то где-то выше, например, в вызывающем методе внешнего объекта) обязан сделать что-то с имеющейся ситуацией. Но не каждая ошибка должна генерировать исключение. Общая рекомендация такова, что исключительными являются ситуации, которые мы не можем обработать в данном контексте. В таком случае необходимо передать управление и ответственность за разрешение этой проблемы обработчику исключений. Например, перед операцией деления можно проверить, не является ли делитель нулем. Но что значит нулевой делитель в данном контексте? Нарушено ли какое-то важное правило бизнес-логики, или это можно как-то локально исправить и пойти дальше?

Когда генерируется (бросается) исключение, на куче создается специального вида объект и управление передается обработчику исключений, назначение которого заключается в обработке подобных ситуаций и восстановлении корректности внутреннего состояния

программы. Например, у нас есть ссылка на некий объект t, и мы хотим убедиться, что она была проинициализирована.

```
if (t == null)
{
    throw new NullReferenceException();
}
```

На самом деле дотнет-машина сама умеет бросать это исключение (как и многие другие), и каждую ссылку проверять не нужно. Если исключение не поймать, программа аварийно завершится (“упадёт”), и пользователю покажут окошко с предложением отправить отчёт об ошибке разработчикам, или что-то вроде, в зависимости от ОС.

У объектов исключений обычно есть 3 конструктора — конструктор по умолчанию, конструктор с одним строковым аргументом, в котором можно передать сообщение, конкретизирующее произошедшее, и конструктор, принимающий помимо сообщения ещё внутреннее исключение. Про внутренние исключения будет немного позже, пока вот пример использования второго конструктора:

```
throw new NullReferenceException("Something is very wrong");
```

Это сообщение, если исключение останется необработанным, потом может быть показано пользователю в каком-нибудь окошке, или, если исключение обработать, записано в лог, или ещё куда-нибудь.

Ключевое слово throw инициирует следующие события:

- создается объект-исключение;
- этот объект неким образом “возвращается” из метода, хотя тип возвращаемого значения метода, естественно, другой, выполнение метода прерывается;
- на стеке ищется метод, содержащий обработчик для данного типа исключений, управление передаётся обработчику, все методы между методом, бросившим исключение, и методом, “поймавшим” исключение, также прерываются.

Так что механизм бросания исключений в некотором смысле можно рассматривать как аналогию механизма возврата значений (однако, на этом сходство заканчивается, т.к. при генерации исключения мы попадаем совсем в другое место, которое может быть очень далеко от места ошибки). throw в каком-то смысле можно понимать как return, вот только использовать его для обычного возврата значений — плохая идея.

В общем-то, бросить можно объект любого класса, унаследованного от класса Exception (корневого класса иерархии исключений .NET), или даже прямо его (`throw new Exception(``Something wrong'');` тоже вполне ок), но хорошим стилем является создание отдельного класса ошибки для каждого типа исключительных ситуаций. Так обработчик исключения по типу и информации, хранящейся в объекте, сможет понять, что же произошло. Собственно, почему так, будет понятно, когда мы рассмотрим, как ловить исключения.

2. Обработка исключений

Для того, чтобы исключения ловить, разберем понятие “охраняемой области” (guarded region). Если бросить исключение внутри метода, то произойдет выход из этого метода. Если же мы хотим остаться внутри этого метода, нужно поместить “опасный код” внутрь блока try. В языках, не поддерживающих механизмы исключений, по сути вы обязаны помещать каждый вызов функции в аналог такого блока (проверка и действия при некорректном значении). Здесь же нужен только один такой блок, это разграничивает основной, “функциональный” код и действия, которые должны осуществляться, если что-то пошло не так.

Блок обработки исключений должен идти сразу же за блоком try:

```
try
{
    // Код, который может бросать исключения
}
catch (Type1 id1)
{
    // Обработка исключения типа Type1
}
catch (Type2 id2)
{
    // Обработка исключения типа Type2
}
catch (Type3 id3)
{
    // Обработка исключения типа Type3
}
finally
{
    // Код, который исполняется всегда, было брошено исключение или нет
}
```

Каждый catch — это как отдельный метод с единственным аргументом, типа ожидаемого исключения. При возникновении исключения обработчики просматриваются по порядку и управление передается первому, тип аргумента которого совпадает с типом объекта исключения (с учетом наследования классов, то есть сам тип и все его наследники подойдут). Из этого, кстати, следует, что типы исключений надо располагать от более частного к более общему, иначе поймается общим обработчиком, а у частного даже не будет шансов (в C# это, кстати, ошибка компиляции). Из этого же следует, что чтобы поймать все исключения вообще, надо ловить по типу Exception, поскольку от него наследуются все остальные (и, разумеется, располагать его в самом низу последовательности catch-ей).

Каждый блок catch можно понимать как функцию, которая принимает параметром объект-исключение указанного типа и ничего не возвращает. Нахождение нужного обработчика путём сопоставления реального объекта-исключения с типами-исключениями, принимаемыми обработчиками, сродни такому понятию функционального программирования, как сопоставление шаблонов (pattern matching). Это, кстати, приводит к тому, что

исключения, определяемые в программе, как правило, никакой содержательной информации, кроме собственно своего типа, не имеют. Тем более, не бывает исключений, имеющих содержательные методы, поэтому пугаться класса, у которого есть один конструктор, и всё, не следует — он нужен лишь как объект для сопоставления с шаблоном в обработчиках.

Блок finally содержит в себе код, который будет исполняться вне зависимости от того, какой обработчик события сработал (и было ли исключение вообще). В языках без сборки мусора блок finally предназначен для того, чтобы закрыть все ресурсы и освободить выделенную память. Память в C# освобождается сборщиком мусора, однако все остальные ресурсы, такие как файлы и сетевые соединения, на вашей совести.

3. Свойства исключений

Объекты, унаследованные от Exception, имеют следующие полезные свойства:

- Data — коллекция пар ключ/значение, в которой может лежать любая дополнительная информация об исключении. На самом деле, не нужна, потому что бросать надо по возможности свои исключения, а в своих исключениях можно определить нормальные поля и свойства с этой самой дополнительной информацией;
- HelpLink — ссылка на страницу с информацией об исключении. Чтобы тот, кто с этим исключением столкнулся, знал, куда смотреть.
- InnerException — информация о предыдущем исключении, которое привело к возникновению текущего. Чуть дальше будет про перебрасывание исключений, будет понятно, что это и зачем.
- Message — текстовое описание исключения.
- Source — информация о месте, где возникло исключение (название сборки или класса). В общем-то не очень полезно, потому что есть более интересное свойство StackTrace.
- StackTrace — распечатка стека вызовов до места возникновения исключения. По нему можно понять, кто кого вызывал и где именно всё упало, очень полезно при отладке.

Небольшой пример:

```
try
{
    throw new Exception("Something is very wrong");
}
catch (Exception e)
{
    Console.WriteLine("Caught Exception");
    Console.WriteLine("e.Message: " + e.Message);
    Console.WriteLine("e.ToString(): " + e.ToString());
    Console.WriteLine("e.StackTrace:\n" + e.StackTrace);
}
```

Вывод в данном случае будет:

```
Caught Exception
e.Message: Something is very wrong
e.ToString(): System.Exception: Something is very wrong
   в CSharpConsoleApplication.Program.Main(String[] args) в c:\Users\yurii_000\D
ocuments\Visual Studio 2012\Projects\CSharpConsoleApplication\CSharpConsoleAppli
cation\Program.cs:строка 15
e.StackTrace:
   в CSharpConsoleApplication.Program.Main(String[] args) в c:\Users\yurii_000\D
ocuments\Visual Studio 2012\Projects\CSharpConsoleApplication\CSharpConsoleAppli
cation\Program.cs:строка 15
Для продолжения нажмите любую клавишу . . .
```

4. Перебрасывание исключений

Допустим, мы в обработчике исключения посмотрели на текущую ситуацию и поняли, что сами мы разобраться с ошибкой не можем и её надо отправить вверх по стеку. Или мы что-то сделали, но всё равно хотим, чтобы исключение попало в обработчик, который выше по стеку. Тогда мы можем, во-первых, бросить то же исключение, во-вторых, бросить новое исключение, при желании передав ему как внутреннее исключение (проперти InnerException) наше исключение, которое мы поймали. Эти способы принципиально отличаются тем, что новый объект-исключение не будет помнить истории старого объекта (StackTrace, например), а если бросить старый объект, то его StackTrace останется неизменным (словно его никто не ловил). Перебрасывать старое исключение, например, так:

```
try
{
    throw new Exception("Something is very wrong");
}
catch (Exception e)
{
    Console.WriteLine("Caught Exception");
    throw;
}
```

Новый объект-исключение можно бросить так:

```
try
{
    throw new Exception("Something is very wrong");
}
catch (Exception e)
{
    Console.WriteLine("Caught Exception");
    throw new Exception("Outer exception", e);
}
```

Ситуация с броском нового объекта кажется довольно экзотичной, но весьма часто используется в дотнетовских библиотеках, так что прежде чем говорить “оно просто упало, и я не могу понять почему”, имеет смысл посмотреть поле InnerException, там может быть написано, что именно пошло не так.

Вообще, в дотнетовской библиотеке куча встроенных классов исключений, про которые можно почитать в документации.

5. Создание своих классов исключений

От Exception наследуются классы SystemException и ApplicationException, считается идеологически правильным наследовать свои классы-исключения именно от ApplicationException, но про это почти никто не знает. Так что в реальных проектах свои исключения, как правило, будут наследовать от Exception. Выглядеть всё это будет примерно так:

```
public class MyException : Exception
{
    public MyException()
    {
    }

    public MyException(string message)
        : base(message)
    {
    }
}
```

Обратите внимание, что исключения довольно часто оказываются вне сборки, в которой объявлены, так что весьма желательно их делать public. Более того, исключения довольно часто оказываются даже не на той машине, на которой были брошены (сейчас клиент-серверных распределённых приложений больше, чем обычных, так что исключение, брошенное сервером, вполне может быть вынужден обрабатывать клиент) так что исключения желательно помечать атрибутом [Serializable], но пока я не рассказывал, что это такое (и, видимо, пока не буду), так что можно не заморачиваться. На самом деле, совсем идеологически правильное объявление своего исключения выглядит так:

```
[Serializable]
public class MyException : Exception
{
    public MyException() { }
    public MyException(string message) : base(message) { }
    public MyException(string message, Exception inner)
            : base(message, inner) { }
    protected MyException(
        System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context)
```

```
        : base(info, context) { }
}
```

Такую штуку студия сгенерит автоматически, если вы наберёте exc и дважды нажмёте Tab.

6. Summary

Итак, исключения можно использовать для того, чтобы:

- устранить проблему и вызвать сфэйливший метод еще раз
- устранить ущерб и отправиться дальше
- предоставить альтернативный результат, который должен был вернуть сфэйливший метод
- сделать все, что можем в текущем контексте и перебросить это исключение наверх
- сделать все, что можем в текущем контексте и бросить наверх другое исключение
- завершить программу
- упрощать код, если ваши схемы обработки исключений все только усложняют, это никуда не годится
- делать ваш код безопаснее, как на короткой перспективе (отладка), так и на длительной (общая надежность приложения)