Глава 7. Концепция M

Это продолжение перевода книги Грегори Декер, Рик де Гроот, Мелисса де Корте. Полное руководство по языку M Power Query. Как и в случае с любым языком программирования, важно понимать как абстрактные принципы языка, так и более практические темы, например, доступные функции и методы их использования. Эта глава призвана дать вам концептуальное представление о ключевых, возможно, более абстрактных аспектах M, таких как область видимости, глобальная среда, замыкание и метаданные. Эти понятия имеют важное значение для того, чтобы по-настоящему овладеть языком M.

> Мои комментарии набраны с отступом.

Предыдущая глава Содержание Следующая глава

Мы начнем с концепции *область*. Понимание области необходимо для управления видимостью и доступностью переменных и функций в запросах. Мы рассмотрим разницу между локальной и глобальной областью видимости, изучив, как переменные и функции взаимодействуют в разных областях.

Обсуждение области видимости естественным образом приводит к понятию *глобальной среды* Power Query, а далее к тому, как мы можем создать собственную глобальную среду с помощью функции *Expression.Evaluate*. Мы рассмотрим различные аспекты глобальной среды и то, как функцию *Expression.Evaluate* можно использовать для создания динамических запросов и программного изменения логики запросов на основе условий или входных данных.

Замыкания являются важной концепцией Power Query, которая расширяет возможности функций. Мы рассмотрим, как замыкания позволяют функциям упаковывать данные и сохранять ссылки на переменные из их лексического окружения, что обеспечивает сложные манипуляции данными и итеративные вычисления.

> Пояснения Chat GPT. Замыкания позволяют функциям сохранять доступ к переменным из внешнего контекста, в котором они были созданы, даже после завершения выполнения этого контекста. Это позволяет функциям *помнить* состояние и работать с ним в будущем.
>
> Лексическое окружение – набор переменных, доступных функции в момент ее создания. Замыкание сохраняет ссылки на эти переменные, а не их значения. Это означает, что функция имеет доступ к этим переменным и может использовать их значения, даже если функция вызывается вне контекста, в котором она была создана. Например:

```
let
  outerFunction = (x) =>
  let
    y = 5,
    innerFunction = () => x + y
  in
    innerFunction
in
  outerFunction
```

> *innerFunction* инкапсулирует (упаковывает) данные *x* и *y*, которые используются для вычисления суммы; *innerFunction* сохраняет ссылки на переменные *x* и *y* из своего лексического окружения (функции *outerFunction*), что позволяет ей использовать их значения при вычислении результата. Даже после завершения выполнения *outerFunction*, *innerFunction* все еще имеет доступ к *x* и *y*. Если вызвать функцию *outerFunction* со значением 10, она вернет 15.

Мы рассмотрим, как использовать метаданные для улучшения управления данными, отслеживания источников данных и обеспечения качества данных.

В этой главе рассматриваются следующие вопросы:

- Общие сведения об области видимости
- Изучение глобальной окружающей среды
- Общие сведения о замыканиях

- Управление метаданными

Для выполнения упражнений необходимо установить Power BI Desktop.

Общие сведения об области видимости

В M область определяет, где можно получить доступ к переменным и функциям и использовать их. Область играет важную роль в управлении потоком преобразований данных.

Глобальная область видимости относится к переменным и функциям, которые определены на уровне запроса, вне какого-либо конкретного шага запроса или функции. Переменные и функции с глобальной областью видимости могут быть доступны и использованы во всех шагах и функциях в пределах всего запроса. Они предоставляют способ хранения и совместного использования данных или вычислений между различными частями запроса, обеспечивая согласованность и возможность повторного использования.

Локальная область – наиболее распространенный тип области в M. Она относится к переменным и функциям, которые определены в определенном шаге запроса или функции. Переменные и функции с локальной областью видимости доступны только в пределах шага или функции, в которой они определены. На них нельзя ссылаться или использовать вне локального контекста, что делает их идеальными для временных вычислений или промежуточных результатов в рамках конкретного преобразования.

Использование выражений let для управления областью видимости: в M выражение *let* позволяет определять локальные переменные в пределах конкретного шага запроса, сохраняя глобальную область видимости чистой и организованной. Это делает код более читабельным и поддерживаемым, разделяя временные вычисления и промежуточные результаты от основных преобразований данных.

Область видимости функции и параметры: функции в Power Query M имеют собственную область видимости. Параметры функции считаются локальными переменными в пределах области видимости функции. Они доступны и используются внутри тела функции, но не имеют значения за ее пределами. Параметры функции позволяют передавать данные и значения между различными частями функции, обеспечивая поток данных и модульность.

Разрешение конфликтов областей: если переменные или функции имеют одно и то же имя в разных областях, Power Query использует процесс, называемый *лексической областью* (также известной как *статическая область*), для разрешения конфликтов областей. Лексическая область действия отдает приоритет локальной области видимости над глобальной, что означает, что если переменная или функция с одинаковым именем существует как локально, так и глобально, локальная переменная будет иметь приоритет в своем конкретном контексте.

Рассмотрим следующий запрос:

```
let
   x = 10,
   y = 20,
   z = 30
in
   x * y * z
```

В следующей таблице приведена сводная информация об области запроса и каждого выражения в запросе:

Элемент	Глобальная среда	Локальная среда
Query	x = 10, y = 20, z = 30	
X		y = 20, z = 30
Y		x = 10, z = 30
Z		x = 10, y = 20

Рис. 7.1. Области действия запроса и отдельных выражений

На уровне запроса все три переменные или выражения (x, y, z) находятся в области действия запроса. Однако каждая переменная имеет собственную локальную область видимости. В этой области каждое выражение может видеть другие глобальные переменные. Однако их

собственное выражение не находится в области видимости и, как правило, на него нельзя ссылаться, не вызывая ошибки. Например, следующий код…

```
let
   x = 10,
   y = 20,
   z = x * y * z
in
   z
```

… вернет ошибку Expression.Error: Имя "z" не распознано. Убедитесь в том, что оно написано верно.

Такой тип ошибки часто является результатом неправильного понимания области видимости. Чтобы предотвратить подобные ошибки, убедитесь, что переменные и функции определены в надлежащей области и доступны там, где вы собираетесь их использовать.

В данном конкретном случае ошибка вызвана выражением z = x * y * z , потому что z не входит в область действия этой строки кода.

Выражение может ссылаться на себя, используя символ @ в качестве префикса. Однако это тоже обычно приводит к ошибке. Например, синтаксически допустим следующий код:

```
let
   x = 10,
   y = 20,
   z = x * y * @z
in
   z
```

Но и он вернет ошибку Expression.Error: Во время оценки обнаружена циклическая ссылка.

Эта ошибка вызвана выражением z = x * y * @z так как движок определил, что нет условия, при котором ссылка на себя завершится. Ссылки на себя полезны и не вызывают ошибок в контексте рекурсивной функции при наличии условия завершения. Об этом подробнее в главе 13 Итерация и рекурсия.

Продолжая пример, покажем, как использование выражения *let* управляет областью видимости, отдавая приоритет локальной области над глобальной:

```
let
  x = 10,
  y =
    let
      x = 1,
      z = x
    in
      z
in
  y
```

Запрос возвращает 1. Как это происходит?

Элемент	Глобальная среда	Локальная среда
Query	x = 10, y = ?	
x		y = ?
y		x = 10
x		x = 10, z = ?
z		x = 1

Рис. 7.2. Области действия запроса и его выражений

Знаком вопроса (?) мы обозначили моменты, которые будут объяснены здесь и в следующих параграфах. Глобальная область действия запроса включает в себя все переменные или

выражения, определенные в запросе (x и y). Каждая из этих переменных имеет локальную область, которая включает в себя все остальные глобальные переменные, но не саму себя.

Выражение переменной *y* включает выражение *let*. Оно управляет областью видимости, в которой мы определяем переменную *x = 1*. Локальная область видимости для этой переменной не может видеть себя, но может видеть глобальную переменную *x = 10*. Локальная область также видит локальную переменную *z*. Переменная *z* не видит себя, но видит *x*. Какую из двух!? Благодаря приоритету локального значения *z* видит *x = 1*. Таким образом, *z* возвращает *1*, и *y* возвращает *1*, и, таким образом, сам запрос возвращает *1*.

Следующий запрос…

```
let
  x = z,
  y = 20,
  z =
    let
      a = 30
    in
      a
in
  x
```

… возвращает 30. Это связано с тем, что *z* находится в области видимости для *x*, и, очевидно, порядок кода не имеет значения. Тот факт, что определение *z* идет после *x*, не имеет значения, поскольку движок M признает, что значение *x* зависит от значения *z*.

И, наконец, запрос…

```
let
  x = a,
  y = 20,
  z =
    let
      a = 30
    in
      a
in
  x
```

… вернет ошибку Expression.Error: Имя "a" не распознано. Убедитесь в том, что оно написано верно.

Это связано с тем, что *a* находится вне видимости *x*. Переменные (выражения), определенные во вложенной структуре *let*, недоступны для выражений, определенных выше по иерархии.

Перейдем теперь к глобальной среде.

Изучение глобальной окружающей среды

В M глобальная среда относится к области верхнего уровня, которая охватывает весь запрос Power Query. Она обеспечивает высочайший уровень видимости переменных и функций, делая их доступными во всем запросе. Глобальная среда служит контейнером для глобальных переменных, функций и параметров, к которым можно получить доступ и использовать их на всех этапах и функциях запроса.

Глобальная среда Power Query включает:

- *Стандартную библиотеку,* представляющую собой неизменяемую коллекцию ресурсов: конфигурационные данные, документация, классы, значения, типы и предварительно написанный код. В стандартную библиотеку M встроены функции, типы и перечисления.
- *Функции расширения с общим доступом,* определяемые с помощью ключевого слова *shared*. Подробнее см. главу 16 Включение расширений.
- *Текущие запросы*, определенные в текущей сессии.

Увидеть актуальную глобальную среду можно с помощью ключевого слова *#shared*. В редакторе Power Query создайте новый пустой запрос, перейдите в расширенный редактор и замените все его содержимое на *#shared*.

#shared является одним из дюжины предопределенных ключевых слов/идентификаторов в языке M, которые начинаются с решетки (#). Большинство ключевых слов/идентификаторов являются конструкторами для типов данных: *#binary*, *#date*, *#time*, ... Есть также *#infinity*, возвращающее символ бесконечности, и *#nan*, возвращающее NaN.

Создав запрос *#shared*, вы получите запись. Преобразуйте ее в таблицу. В ней вы найдете все запросы, определенные в сеансе и все стандартные функции, типы и перечисления M, а также функции сторонних коннекторов, которые поставляются в комплекте с Power BI Desktop.

	Name	Value
1	Запрос1	Table
2	Вызванная функция	Error
3	Запрос2	Table
4	Value.ResourceExpression	Function
5	Resource.Access	Function
6	Kusto.Contents	Function
7	Kusto.Databases	Function
8	AzureDataExplorer.Contents	Function
9	AzureDataExplorer.Databases	Function
10	AzureDataExplorer.KqlDatabase	Function
11	CommonDataService.Database	Function
12	PowerPlatform.Dataflows	Function
13	DataLake.Contents	Function
14	DataLake.Files	Function
15	Fabric.Warehouse	Function

Рис. 7.3. Глобальная среда

Прежде чем мы продолжим, давайте рассмотрим две дополнительные темы, связанные с глобальной средой.

Разделы

Более подробно разделы рассматриваются в главе 16. Здесь мы объясним общую концепцию. Разделы в Power Query являются контейнерами для различных элементов, таких как запросы, параметры, функции, выражения и др. Эти элементы называются членами. Разделы нельзя создать в редакторе Power Query, их можно реализовать как расширения глобальной среды.

Хотя разделы не могут быть созданы в редакторе Power Query, мы можем просмотреть разделы, доступные в глобальной среде, с помощью ключевого слова *#sections*. Создайте пустой запрос, откройте расширенный редактор, замените все содержимое запроса ключевым словом: *#sections*. Возвращается запись с одним полем *Section1* – разделом по умолчанию, автоматически созданном Power Query. Измените запрос:

```
#sections[Section1]
```

```
= #sections[Section1]
```

Запрос1	Table
Вызванная функци	Error
Запрос2	Table
Запрос3	Record

Рис. 7.4. Содержимое раздела *Section1*

Возвращается запись с полями для каждого запроса текущего сеанса редактора Power Query. Мы можем получить доступ к любому запросу, используя синтаксис:

```
#sections[Section1][Запрос1]
```

Этот код возвращает результаты *Запрос1*. Более предпочтительный синтаксис для достижения той же цели:

Section1 ! Запрос1

Мы напрямую ссылаемся на *Section1*, без ключевого слова *#sections*, а восклицательный знак (!) используется в качестве сокращения для доступа к *Запрос1*.

Для каждого сеанса редактора Power Query глобальная среда содержит один раздел (*Section1*), включающий все запросы в качестве членов раздела. Такая иерархия позволяет ссылаться на запрос из другого запроса. Она позволяет создавать промежуточные запросы, которые не загружаются в семантическую модель. Мы рассмотрели один пример в главе 3, когда загружали все файлы из папки,

Если у вас есть несколько разделов, то выражение *#sections* внутри раздела вернет иные результаты. Вместо одного раздела (*Section1*) выражение вернет свой раздел для каждого расширения, включенного в Power BI Desktop, а также любого пользовательского расширения. Точно так же, запуск выражения *#shared* внутри раздела вернет результаты отличные от выполнения *#shared* в стандартной глобальной среде. Возвращаемая запись будет содержать только выражения, помеченные ключевым словом *shared* внутри раздела, а также все стандартные библиотечные функции, типы и перечисления M.

Кратко рассмотрев разделы, перейдем к тому, как мы можем создать собственную глобальную среду для контроля переменных и функций, доступных в выражении.

Создание собственной глобальной среды

Глобальная среда, которую мы только что рассмотрели, является стандартной глобальной средой. Можно создать собственную глобальную среду с помощью функции *Expression.Evaluate*.

Функция *Expression.Evaluate* имеет два параметра: вычисляемое M-выражение и необязательный параметр среды в виде записи. Он описывает среду, в которой вычисляется выражение M. По умолчанию он имеет значение *null*. Таким образом, следующий запрос возвращает нулевую запись:

```
Expression.Evaluate("#shared")
```

Так как это пустая глобальная среда, в ней также нет разделов. Мы можем убедиться в этом с помощью кода…

```
Expression.Evaluate("#sections")
```

… который также вернет нулевая запись.

Если мы добавим запись среды…

```
Expression.Evaluate("#shared", [x = 10, y = 20, z = 30])
```

… она вернет запись:

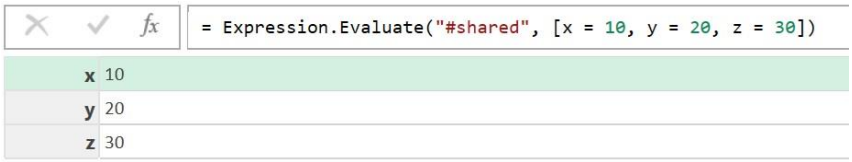

Рис. 7.5. Пример собственной глобальной среды

Вы можете передать в среду функции. Например:

```
Expression.Evaluate(
  "let Source = List.Sum( {x, y, z} ) in Source",
  [x = 10, y = 20, z = 30, List.Sum = List.Sum]
) // возвращает 60
```

Вы даже можете использовать *Expression.Evaluate* для переименования стандартных функций M в вашей среде. Следующий код также вернет 60:

```
Expression.Evaluate(
  "let Source = SumThatList( {x, y, z} ) in Source",
  [x = 10, y = 20, z = 30, SumThatList = List.Sum]
```

)

Выполнение выражений в собственной среде, состоящей только из элементов, необходимых для вычисления, можно считать мерой безопасности. Если вас не волнуют такие вещи, передайте *#shared* в *Expression.Evaluate* в качестве второго аргумента, и вычисления пройдут в стандартной глобальной среде. Следующее выражение является допустимым и вернет 60:

```
Expression.Evaluate(
   "let x = 10, y = 20, z = 30 in List.Sum( { x, y, z } )",
   #shared
)
```

Перейдем к важной, но довольно абстрактной теме – замыканиям. Понимание замыкания поможет вам создавать динамический, многократно используемый код в языке M.

Общие сведения о замыканиях

В языках программирования замыкание – это мощная концепция, которая позволяет функции захватывать и сохранять ссылки на переменные из своей лексической среды (среды, в которой определена функция). Это означает, что даже после того, как внешняя функция завершила выполнение или вышла за пределы области видимости, внутренняя функция (замыкание) по-прежнему сохраняет доступ к переменным из окружающей области.

Замыкания создаются, когда внутренняя функция ссылается на переменные из содержащей ее функции или любой другой окружающей области. Внутренняя функция замыкается над этими переменными, отсюда и термин замыкание.

Способность замыкания сохранять доступ к переменным из своего лексического окружения особенно полезна в сценариях, где необходимо создать функции с поведением, зависящим от значений некоторых переменных на момент определения функции. Например:

```
let
  x = 10,
  closureFunction = () => x * 2
in
  closureFunction() // возвращает 20
```

Переменная *x* определена в блоке *let* и имеет значение 10 (глобальная область). *closureFunction* является замыканием, так как захватывает ссылку на переменную *x* из ее лексического (текущего) окружения. Когда *closureFunction* вызывается позже в запросе, она по-прежнему имеет доступ к значению *x = 10* и вернет результат *x * 2 = 20*.

Рассмотрим более сложный пример замыкания:

```
let
  x = 10,
  closureFunction = () => x * 2,
  Evaluation = Expression.Evaluate("closureFunction()", [closureFunction = closureFunction, x = 20])
in
  Evaluation
```

Учитывая наше объяснение приоритета локальной области, можно ожидать, что запрос вернет 40 (20 * 2). Однако это не так. На самом деле это выражение возвращает 20 (10 * 2). Неужели мы просто выбросили в окно все наши обсуждения области действия и приоритеты локальной области? Вовсе нет, и объяснение – замыкание.

Обратите внимание, что *closureFunction* изначально определена в области, где *x = 10*. Как и в исходном примере, *closureFunction* по-прежнему захватывает ссылку на переменную *x* из ее исходного лексического (текущего) окружения. Таким образом, это замыкание захвата текущих значений в контексте того, где функция изначально определена, имеет приоритет над передачей другого значения *x* (20) в среду выражения *Expression.Evaluate*, где мы вызываем *closureFunction*. Другими словами, значение *x* из области видимости, в которой функция была первоначально определена, является *закрытым* и не может быть изменено.

Сделаем еще шаг в концепции исходного контекста. Рассмотрим запрос:

Запрос 7.1

```
let
   multiplyFunction = ( x ) as function => ( multiplier ) => x * multiplier,
   closureFunction = multiplyFunction( 10 )
in
   closureFunction( 2 ) // вернет 20
```

Как это работает? На первый взгляд может показаться, что код выдаст ошибку, поскольку ни в какой момент времени, *x* и *multiplier* не находятся в одной и той же области видимости. Секрет в создании функции, возвращающей функцию, и применении замыкания. Функция *multiplyFunction* принимает *x*, и возвращает функцию, которая принимает один параметр, *multiplier*. Мы определяем *closureFunction* как функцию, которая возвращается при вызове *multiplyFunction* со значением 10.

Замыкание вступает в игру, потому что *closureFunction* запоминает свой исходный контекст, в котором для *x* установлено значение 10, путем вызова *multiplyFunction* с аргументом 10. Таким образом, определение функции, возвращаемой *multiplyFunction*, закрывается и определяется как функция, где *x = 10*. Когда позже мы вызываем *closureFunction* со значением 2, передаваемый параметр является параметром множителя функции, возвращенной при вызове *multiplyFunction* со значением 10, и, следовательно, 10 * 2 = 20.

Замыкания полезны для создания многократно используемых функций с динамическим поведением. Они позволяют определять функции, зависящие от значений переданных позже за пределами тела функции, что обеспечивает гибкость и адаптируемость кода. Замыкания используются в сценариях, где необходимо создавать функции с поведением на основе контекста или условий. Замыкания также необходимы для реализации свертывания запросов.

Свертывание запросов

Свертывание запросов (*query folding*) – это функция Power Query, которая требует определенной логики реализации в коннекторах с помощью функции *Table.View*. Свертывание запроса происходит, когда код M переписывается в запрос к источнику данных. Как правило, свертывание запросов повышает производительность Power Query. Свертывание запросов передает часть преобразований на сторону сервера. Мы подробно рассмотрим свертывание запросов в главе 15 Оптимизация производительности.

Важно понимать, что сам язык M ничего не знает о свертывании запросов. Это означает, что свертывание запросов не является частью ядра M или его стандартной библиотеки. По сути, при выполнении кода M в Power Query процесс перехватывается, а запрос переписывается с использованием обработчиков событий в коннекторе, если такие обработчики событий существуют. Другими словами, вместо того, чтобы обработчик M выполнял преобразования запросов на своей стороне, код переписывается и выполняется исходной системой, а не M.

Обработчики событий запускаются при наступлении определенного события, например при запросе таблицы базы данных SQL Server. Событие запускает обработчик событий, который затем выполняет задачу. В случае свертывания запросов обработчик событий отвечает за переписывание запроса M в запрос к исходной системе. Поскольку свертывание запросов не является частью языка M, мы рассмотрим эту концепцию кратко.

Для свертывания запроса коннектор должен реализовать функцию, которая возвращает объект *Table.View*. *Table.View* включает определение обработчика событий, переданные в виде записи. Существует два типа обработчиков событий: *Get* и *On*. Класс обработчиков событий *On* обрабатывает захват, когда запрашивается определенное событие, такое как *OnSort*, *OnTake*, *OnSkip* или *OnSelectColumns*. Класс обработчиков событий *Get* возвращает данные или информацию о данных: *GetType*, *GetRows* и *GetRowCount*. Следующий запрос M извлекает таблицу *DimCurrency* из базы данных SQL Server и сортирует ее по столбцу *CurrencyAlternateKey*:

```
let
  Source         = Sql.Database("localhost", "AdventureWorksDW2022"),
  dbo_DimCurrency = Source{[Schema = "dbo", Item = "DimCurrency"]}[Data],
```

```
  #"Sorted Rows"  = Table.Sort(dbo_DimCurrency, {{"CurrencyAlternateKey", Order.Ascending}})
in
  #"Sorted Rows"
```

Так как коннектор для SQL Server поддерживает свертывание запросов, а также обработчики событий *GetRows* и *OnSort*, этот запрос переписывается как следующий SQL-запрос:

```
select [_].[CurrencyKey],
   [_].[CurrencyAlternateKey],
   [_].[CurrencyName]
from [dbo].[DimCurrency] as [_]
order by [_].[CurrencyAlternateKey]
```

SQL-запрос можно увидеть в редакторе Power Query, щелкнув правой кнопкой мыши последний шаг в области *Примененные шаги* и выбрав *Просмотреть машинный запрос*.

Хотя очевидно, **что** возвращает класс обработчиков событий *Get*, но что же возвращает класс обработчиков событий *On*? Класс обработчиков событий *On* выполняет замыкания для захвата информации, относящейся к запросу на свертывание, и возвращает объект *Table.View*, который запоминает эту информацию. Таким образом, когда обработчик событий класса *Get* получает доступ к *Table.View*, возвращенному из обработчика событий класса *On*, *Table.View* уже закодировал детали запроса на свертывание.

Этот механизм подобен тому, что и в запросе 7.1, где *closureFunction* запоминает , что значение *x = 10*. Подробнее о свертывания запросов можно узнать из официальной документации.

> См. также Язык M Power Query. Свертывание запросов.

Перейдем к концепции метаданных.

Управление метаданными

Метаданные – это данные о данных. В Power Query M метаданные можно связать с любым значением с помощью ключевого слова *meta* для определения записи метаданных. Метаданные сами по себе ничего не делают и никак не изменяют поведение значения.

```
let
  x = 10 meta [Type = "Whole number", OoM = 1],
  y = 20 meta [Type = "Whole number", OoM = 1],
  z = 30 meta [Type = "Whole number", OoM = 1]
in
  x * y * z
```

Этот запрос возвращает 6000 несмотря на метаданные. Чтобы вернуть метаданные, связанные с каждым значением, используйте функцию *Value.Metadata*:

```
let
  x = 10 meta [Type = "Whole Number", OoM = 1],
  y = 20 meta [Type = "Whole Number", OoM = 1],
  z = 30 meta [Type = "Whole Number", OoM = 1]
in
  Value.Metadata(x)[Type]
```

Этот запрос возвращает Whole Number (целое число).

Чтобы очистить значение от метаданных используйте функцию *Value.RemoveMetadata*:

```
let
  x  = 10 meta [Type = "Whole Number", OrderOfMagnitude = 1],
  x1 = Value.RemoveMetadata(x)
in
  Value.Metadata(x1)
```

Чтобы заменить метаданные новой записью метаданных используйте *Value.ReplaceMetadata*:

```
let
```

```
  x = 10 meta [Type = "Whole Number", OrderOfMagnitude = 1],
  x1 = Value.ReplaceMetadata(x, [Divisors = 4])
in
  Value.Metadata(x1)
```

Этот запрос возвращает запись с одним полем, *Divisors*, со значением 4. А вот следующий код добавляет новые метаданные к исходным метаданным:

```
let
  x = 10 meta [Type = "Whole Number", OrderOfMagnitude = 1],
  x1 = Value.ReplaceMetadata(x, Value.Metadata(x) & [Divisors = 4])
in
  Value.Metadata(x1)
```

Этот запрос возвращает запись с тремя полями: *Type*, *OoM* и *Divisors*.

Наконец, рассмотрим следующий пример:

```
let
  x = 10 meta [Type = "Whole Number", OrderOfMagnitude = 1],
  x1 = Value.ReplaceMetadata(x, Value.Metadata(x) & [Divisors = 4]),
  multiply = x * x1
in
  Value.Metadata(multiply)
```

Что вернет запрос? Метаданные, прописанные для *x* или для *x1*? Ни один из них! На выходе будет нулевая запись. Это связано с тем, что метаданные не переносятся на результат операции.

Как уже говорилось, метаданные сами по себе ничего не делают, и нет никаких специальных значений метаданных, распознаваемых движком M. Однако приложения (в том числе редактор Power Query) используют метаданные для изменения своего поведения.

Например при клике на кнопку *Получить данные* в диалоговом окне *Навигатор* метаданные управляют значками, отображаемыми для элементов навигации, а также тем, является ли элемент папкой или конечным источником, возвращающим данные предварительного просмотра. Более подробную информацию по этой теме можно найти в главе 16 Включение расширений.

Еще один пример – создание запроса, использующего функцию *Value.Metadata* для другого запроса. При этом возвращается запись с полем *QueryFolding*, значением которого является запись. При развертывании этой записи возвращаются поля (со значениями): *IsFolded, HasNativeQuery, Kind, Path*.

И наконец, метаданные в редакторе Power Query предоставляют документацию по функциям. Создайте пустой запрос, откройте расширенный редактор, и замените все содержимое запроса строкой *Table.AddColumn*. Отобразится следующая информация:

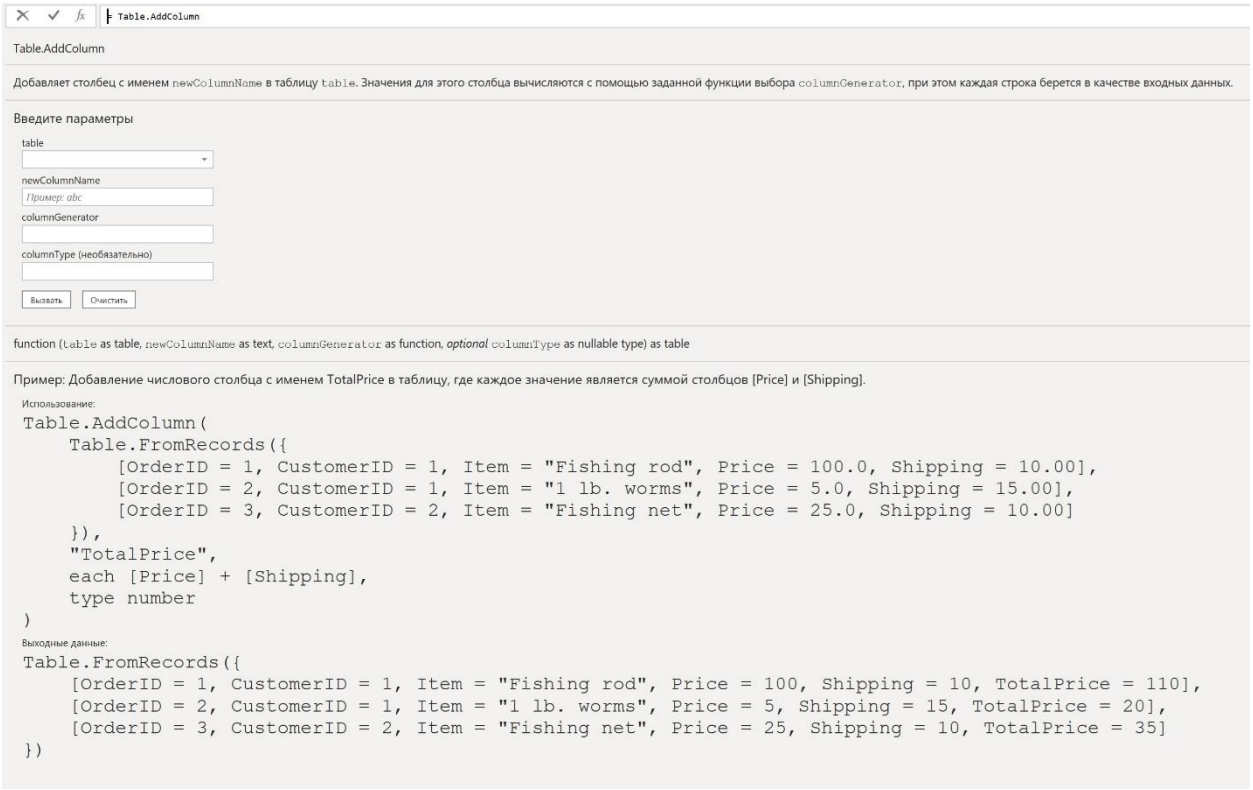

Рис. 7.6. Внутренняя документация для функции *Table.AddColumn*

Сравните это с выходными данными, возвращаемыми пользовательской функцией с названием *multiplyFunction* и кодом:

```
(multiplier as number) as number =>
  let
    x = 10
  in
    x * multiplier
```

В этом случае информации гораздо меньше. Есть только ссылка на *multiplier*:

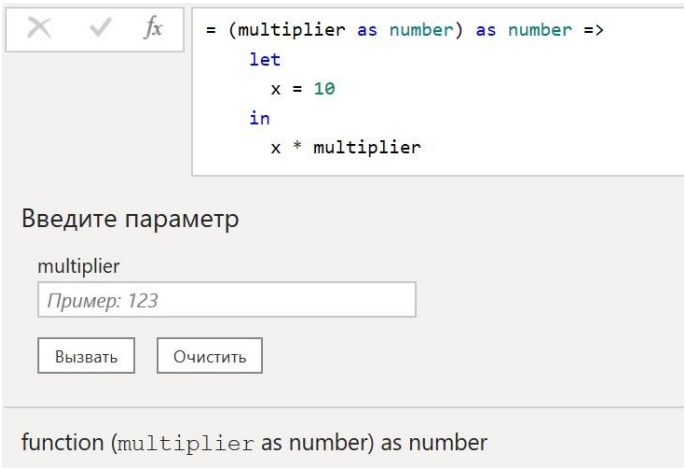

Рис. 7.7. Документация по пользовательской функции

Мы можем добавить метаданные, чтобы документация по пользовательской функции стала интереснее:

```
let
 Source = (multiplier as number) as number =>
  let
    x = 10
  in
```

```
    x * multiplier,
 Type = type function (value as number) as number
  meta [
    Documentation.Name = "multiplyFunction",
    Documentation.LongDescription = "Multiplies the number 10 by the multiplier.",
    Documentation.Examples = {
      [Description = "Multiply by 1", Code = "multiplyFunction(1)", Result = "10"],
      [Description = "Multiply by 2", Code = "multiplyFunction(2)", Result = "20"],
      [Description = "Multiply by 3", Code = "multiplyFunction(3)", Result = "30"]
    }
  ]
in
 Value.ReplaceType(Source, Type)
```

В результате получится:

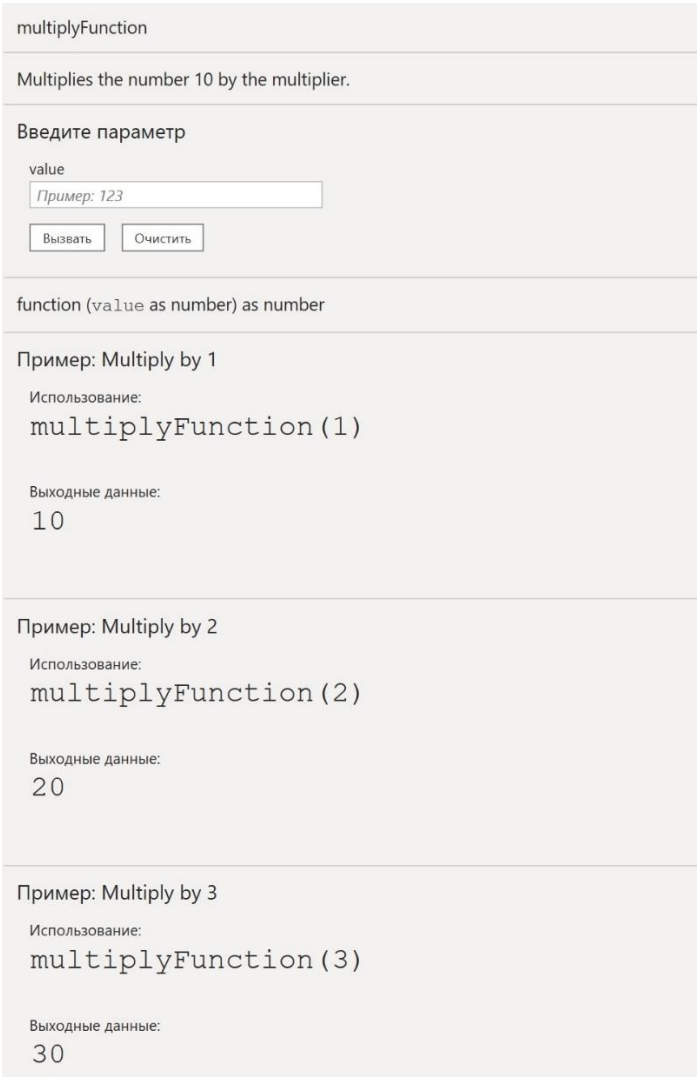

Рис. 7.8. Расширенная документация по пользовательской функции

Поля записей метаданных, используемые в этом примере, являются специальными полями, распознаваемыми редактором Power Query. Эти поля записи метаданных должны быть прикреплены к типу функции, а не к самой функции. Поэтому необходимо создать новый тип функции с нужной записью метаданных, а затем связать этот тип с нашей функцией через функцию *Value.ReplaceType*.

Существуют и другие поля записей метаданных. *Documentation.Description* изменяет описание функции. Эта запись метаданных будет переопределена, если также существует *Documentation.LongDescription*. *Documentation.Syntax* и *Documentation.Result* не оказывают никакого влияния на пользовательский интерфейс Power Query.

Существуют специальные поля записи метаданных, которые можно использовать для параметров функций. Вот пример использования записи метаданных для параметра *multiplier*:

```
let
  Source = (multiplier as number) as number =>
    let
      x = 10
    in
      x * multiplier,
  Type = type function (
    multiplier as (
      type number
        meta [
          Documentation.FieldCaption  = "Multiplier – The multiplier of 10",
          Documentation.AllowedValues = {1 .. 10}
        ]
    )
  ) as number
    meta [
      Documentation.Name = "multiplyFunction",
      Documentation.LongDescription = "Multiplies the number 10 by the multiplier.",
      Documentation.Examples = {
        [Description = "Multiply by 1", Code = "multiplyFunction(1)", Result = "10"],
        [Description = "Multiply by 2", Code = "multiplyFunction(2)", Result = "20"],
        [Description = "Multiply by 3", Code = "multiplyFunction(3)", Result = "30"]
      }
    ]
in
  Value.ReplaceType(Source, Type)
```

Здесь мы связываем запись метаданных с определением типа для параметра *multiplier*. *Documentation.FieldCaption* заменяет заголовок параметра по умолчанию, а *Documentation.AllowedValues* изменяет поле ввода на раскрывающийся список указанных значений. На самом деле это не изменяет значения, которые могут быть переданы в функцию, это влияет только на интерфейс в редакторе Power Query:

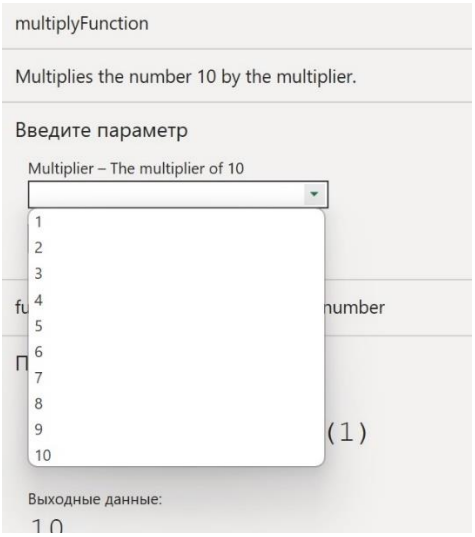

Рис. 7.9. Расширенная документация по параметрам пользовательской функции

Дополнительную информацию о функциях документирования можно найти на официальном сайте.

Большинство пользователей языка M редко применяют метаданные. Это печально, так как метаданные представляют собой огромную область неиспользованного потенциала, которая

может позволить делать в коде M вещи, которые в противном случае были бы невозможны. Например, рассмотрим запрос *PreviousSteps*:

```
let
  Step1  = 10,
  Step2  = Step1 * 2,
  Step2a = Step2 meta [PreviousStep = Step1],
  Step3  = Step2 * 2,
  Step3a = Step3 meta [PreviousStep = Step2a]
in
  Step3a
```

Этот запрос возвращает 40. Если мы создадим другой запрос, который ссылается на этот запрос, то он также вернет 40. Как правило, другим запросам доступны только выходные данные последнего шага запроса. Однако, поскольку мы добавили метаданные, мы можем получить доступ к значениям предыдущих шагов в запросе. Следующий запрос возвращает значение 20 из шага 2:

```
let
  Source = Value.Metadata(PreviousSteps)[PreviousStep]
in
  Source
```

В отличие от этого, следующий запрос возвращает 10 из Step1:

```
let
  Source = Value.Metadata(Value.Metadata(PreviousSteps)[PreviousStep])[PreviousStep]
in
  Source
```

На этом мы завершаем изучение метаданных в языке M. Как уже упоминалось, метаданные являются, пожалуй, самым большим источником неиспользованного потенциала языка M, поэтому помните о метаданных, продолжая расширять свои практические знания M.

> См. также Язык M Power Query. Метаданные.

Саммари

В этой главе мы рассмотрели важные, но скрытые и абстрактные аспекты языка M, включая область видимости, глобальное окружение, замыкания и метаданные. Мы также кратко изучили разделы, и создание собственного глобального окружения с помощью функции *Expression.Evaluate*. Разделы будут рассмотрены более подробно в главе 16 Включение расширений.

Хотя информация в этой главе может показаться не столь практичной как в других главах, представленные концепции критически важны для полного понимания языка M и его потенциала.

В следующей главе мы начнем применять абстрактные концепции из этой главы на практике, работая с вложенными структурами, где концепция области видимости имеет жизненно важное значение.