Erlang: Практикум

Чтобы накопленные знания не заржавели, **Андрей Ушаков** устраивает пробежку по реальным примерам и задачам.

Наш эксперт

Андрей Ушаков активно приближает тот день, когда функциональные языки станут мейнстримом.

В прошлом номере журнала (**LXF150**) мы закончили рассмотрение базовых сущностей языка Erlang. И, как часто бывает, далеко не просто применить полученные знания на практике, особенно с учетом того факта, что концепции функционального программирования достаточно сильно отличаются от концепций императивного, с которыми знакомы большинство программистов. Поэтому, прежде чем идти дальше, было бы полезно рассмотреть все, что мы уже изучили на практике, на реальных задачах.

А начнем мы с того, как не надо объявлять функции. На первый взгляд, объявить функцию достаточно просто, и никаких подводных камней при этом быть не может. Но это только на первый взгляд: некорректное объявление функции может принести немало сюрпризов, а ее правильное использование – очень сильно упростить код. Мы помним (см. **LXF145**), что при объявлении функции допускается объявить несколько ее вариантов. Выбор варианта, который будет использован, осуществляется во время вызова функции; при этом на выбор варианта влияют два механизма: соответствие шаблону [pattern-matching] и выражения охраны [guards]. И неудивительно, что при неаккуратном использовании операции соответствия шаблону и выражений охраны мы можем получить функцию, у которой один или несколько вариантов никогда не будут выбраны, либо будут выбраны не те варианты, которые ожидались.

Давайте рассмотрим на примерах, как такое может получиться (и, соответственно, как нам не стоит делать). В первом примере показано неправильное использование операции соответствия шаблону, когда вариант функции с общим выражением соответствия шаблону идет раньше вариантов с более конкретными выражениями.

```
test([1 | _Other]) -> one;
test([1, 1 | _Other]) -> eleven;
test([1, 0 | _Other]) -> ten;
test(_Other) -> other.
```

В этом объявлении первый вариант функции **test/1** является общим по отношению ко второму и третьему вариантам (т. к. первый вариант ожидает список, начинающийся с 1, второй вариант ожидает список, начинающийся последовательно с 1 и 1, а третий вариант ожидает список, начинающийся последовательно с 1 и 0). Поэтому при вызовах **test([1, 2])**, **test([1, 0])** и **test([1, 1])** всегда будет выбран первый вариант, и результатом этих вызовов будет атом **one**. Заметим, что компилятор данную ситуацию понимает и генерирует предупреждение. В следующем примере показана ситуация, которую компилятор уже не понимает и, соответственно, никаких предупреждений не выдает.

```
test(X) when is_number(X) -> number;
test(X) when X < 0 -> negative;
test(0) -> zero;
test(_Other) -> other.
```

Здесь мы объявляем функцию с несколькими вариантами, которые различаются как при помощи выражений охраны (первый и второй варианты), так и при помощи соответствия шаблону (третий вариант). Первый вариант проверяет, является ли аргумент функции числом, второй вариант – меньше ли аргумент нуля, третий вариант – равен ли аргумент нулю. Очевидно, что первый вариант будет выполняться всегда, когда будут выполняться второй и третий варианты, поэтому вызовы **test(1)**, **test(-1)** и **test(0)** всегда вернут атом **number**. На второй вариант функции **test/1** хочется обратить особое внимание: в этом варианте в выражении охраны мы просто проверяем, меньше ли аргумент функции 0. Мы помним (см. **LXF150**), что в языке Erlang позволено сравнивать данные разных типов и что числа всегда меньше объектов других типов. Поэтому второй вариант функции **test/1** никогда не будет выбран, если передавать в аргументе функции объект другого типа.

Давайте поменяем второй вариант функции **test/1** следующим образом:

```
test(X) when X > 0 -> positive;
```

Эффект такой замены будет противоположным: для всех объектов не числовых типов, передаваемых в качестве аргумента функции **test/1**, будет выбран именно этот вариант (а для всех чисел будет выбран всегда первый вариант). И картина еще более усложнится, если мы одновременно будем использовать и выражения охраны, и соответствия шаблону, либо составные выражения. Мораль всего этого такова: будьте крайне внимательны к условиям выбора того или иного варианта функции и старайтесь создавать варианты функций, условия выбора которых как можно проще (пускай даже за счет увеличения количества вариантов). И последнее, что можно сказать на основании этих примеров. В качестве последнего варианта функции **test/1** мы задавали тот, который выбирался, если все другие варианты не подходили. В реальных задачах такой вариант вряд ли будет нужен. Конечно, если мы опустим такой общий вариант и не будет выбран ни один из более конкретных вариантов, то мы получим ошибку времени выполнения. Но с другой стороны, возникновение такой ошибки означает, что у нас произошло нарушение контракта и состояние системы стало неопределенным, а в такой ситуации генерация ошибки является единственно правильным вариантом.

Пойдем дальше и рассмотрим еще один пример на объявление функции с несколькими вариантами. Мы только что рассмотрели, как не надо объявлять варианты функции и к каким «граблям» может привести неправильное объявление. В этом примере мы увидим, что грамотное использование объявления функции с несколькими вариантами делает код намного понятнее и компактнее. Итак, наша задача – создать функцию для форматирования даты и времени в соответствии со строкой формата. В строке формата следующие символы (без двойных кавычек) заменяются соответствующими значениями: **“DD”** – день, **“MM”** – месяц, **“YY”** – последние две цифры года, **“YYYY”** – год, **“hh”** – час, **“mm”** – минуты, **“ss”** – секунды, **“ms”** – миллисекунды. Все остальные символы остаются без изменений. Для удобства работы с датой и временем (в языке Erlang нет специального типа для даты и времени) объявим следующий тип записи (см. **LXF146**):

```
-record(datetime, {day = 0, month = 0, year = 0, hour = 0, minutes =
0, seconds = 0, milliseconds = 0}).
```

Далее определим интерфейсную функцию **format/2**, т. е. функцию с меньшим числом параметров, которая вызывает функцию,

ПО СУЩНОСТЯМ

реализующую интересующую нас функциональность, и передает ей необходимое число параметров. Это общий подход сокрытия деталей реализации: создаются интерфейсная и реализующая функции, интерфейсная функция принимает только необходимые параметры и вызывает реализующую функцию, передавая ей как необходимые, так и вспомогательные (для выполнения алгоритма) параметры.

```
format(FormatString, DateTime) -> format(FormatString, DateTime, “”).
```

А теперь определим функцию, реализующую основную функциональность форматирования даты **format/3**. От интерфейсной функции **format/2** эта функция отличается одним лишним параметром – приемником для результирующей строки. В реализующей функции за один вызов мы обрабатываем 1, 2 или 4 символа, пока все символы в строке формата не будут обработаны. Чтобы определить, сколько символов за один вызов необходимо обработать, создадим несколько вариантов функции **format/3**. Рассмотрим эти варианты подробно. Первый определяет случай, когда мы рассмотрели строку формата полностью. Тогда мы должны вернуть результирующую строку; но мы помним (см. **LXF147**), что эффективнее добавлять новые элементы в начало списка, поэтому перед возвратом результирующую строку необходимо перевернуть (при помощи функции **lists:reverse/1**).

```
format([], _DateTime, Dest) -> lists:reverse(Dest);
```

Следующий вариант определяет случай, когда необходимо заменить два символа из строки формата значением дня даты в результирующей строке (когда два первых символа в остатке строки формата – **“DD”**). В этом варианте мы рекурсивно (при помощи хвостовой рекурсии) вызываем сами себя (функцию **format/3**), в строки формата мы передаем остаток от входной строки формата (без двух символов), в качестве результирующей строки – входную результирующую строку, к которой добавлено значение дня даты (в начало строки в обратном порядке).

```
format([$D, $D | Rest], DateTime, Dest) -> format(Rest, DateTime, integer_to_rstring(DateTime#datetime.day, 2) ++ Dest);
```

Следующий вариант аналогичен предыдущему, только по отношению к значению месяца даты.

```
format([$M, $M | Rest], DateTime, Dest) -> format(Rest, DateTime, integer_to_rstring(DateTime#datetime.month, 2) ++ Dest);
```

В следующих двух вариантах мы обрабатываем ситуацию, когда в результирующую строку необходимо подставить значение года даты вместо спецификатора в строке формата. Первый вариант обрабатывает ситуацию, когда спецификатор в строке формата **“YYYY”**, а второй вариант – когда спецификатор **“YY”**. Эти два варианта специально идут в таком порядке: если их поменять местами, то всегда будет выбираться (и при спецификаторе **“YY”**, и при спецификаторе **“YYYY”**) вариант, обрабатывающий короткий спецификатор года (как уже говорилось выше, компилятор в этом случае нас предупредит).

```
format([$Y, $Y, $Y, $Y | Rest], DateTime, Dest) -> format(Rest, DateTime, integer_to_rstring(DateTime#datetime.year, 4) ++ Dest);
format([$Y, $Y | Rest], DateTime, Dest) -> format(Rest, DateTime, integer_to_rstring(DateTime#datetime.year rem 100, 2) ++ Dest);
```

В следующих четырех вариантах мы обрабатываем ситуации, когда первые два символа являются спецификаторами значения часа, минут, секунд, миллисекунд соответственно.

```
format([$h, $h | Rest], DateTime, Dest) -> format(Rest, DateTime, integer_to_rstring(DateTime#datetime.hour, 2) ++ Dest);
format([$m, $m | Rest], DateTime, Dest) -> format(Rest, DateTime, integer_to_rstring(DateTime#datetime.minutes, 2) ++ Dest);
format([$s, $s | Rest], DateTime, Dest) -> format(Rest, DateTime, integer_to_rstring(DateTime#datetime.seconds, 2) ++ Dest);
format([$m, $s | Rest], DateTime, Dest) -> format(Rest, DateTime, integer_to_rstring(DateTime#datetime.milliseconds, 3) ++ Dest);
```

И наконец, последний вариант обрабатывает ситуацию, когда первые 2 или 4 символа не являются ни одним из спецификаторов формата. В этом случае мы просто копируем первый символ из остатка строки формата в результирующую строку. Следует заметить, что в этом варианте (который обрабатывает все оставшиеся ситуации) мы проверяем при помощи выражения охраны, является ли первый элемент списка (строки формата) символом в кодировке **ISO-latin-1 (ISO8859-1)**. Если это не так, то будет сгенерировано исключение.

```
format([Char | Rest], DateTime, Dest) when is_integer(Char), Char > 0, Char < 256 -> format(Rest, DateTime, [Char] ++ Dest).
```

Вот и все варианты функции **format/3**. Так как функция **format/3** является рекурсивной, то следует сделать следующее замечание: при каждом рекурсивном вызове мы в итоге получаем (и передаем дальше) остаток строки формата минимум на один символ меньше, чем он был на вход. Это означает, что наша рекурсивная обработка когда-нибудь завершится, что не может не радовать.

Теперь нам осталось рассмотреть вспомогательную функцию **integer_to_rstring/2.** Эта функция переводит целое число в строку заданной длины (заполняя ее символами **“0”** слева, если »

Коротко о записях

Записи – это кортежи, организованные специальным образом: первым элементом такого кортежа идет имя записи (имя записи является атомом), после которого идут значения полей в порядке их объявления. При этом мы можем задавать значения полей в любом порядке, а компилятор расположит их в правильном порядке. При создании записи те поля, для которых значение не было задано, получат значение по умолчанию. Если значение по умолчанию для поля не определено, то значением по умолчанию становится атом **undefined**. В общем виде определение записи имеет следующий вид: **-record(Name, {Field1 [= Value1], ..., FieldN [= ValueN]})**. После определения создать экземпляр записи можно так: **#Name{Field1=Expr1,...,FieldK=ExprK}**.

Как правильно использовать записи для взаимодействия с кодом из других модулей? Есть несколько вариантов это сделать:

» Задать определение записи в отдельном файле (обычно с расширением **.hrl**) и подключать этот файл при помощи директивы препроцессора **–include(...)**, где это необходимо.

» В консоли среды **Erlang** (при использовании записей в консоли среды Erlang их нельзя подключить, как в обычном коде) определить запись при помощи директивы **rd**: **rd(Name, {Field1 [= Value1], ..., FieldN [= ValueN]})**.

» Вместо записи использовать кортеж, но при этом необходимо как соблюдать порядок полей (использованный при описании записи), так и не пропускать поля со значениями по умолчанию: **{Name, Expr1,...,ExprK}**. Следует сказать, что в реальном коде так поступать не следует (т. к. при изменении структуры записи использование кортежа будет нарушением контракта), но для целей тестирования (например, в консоли среды **Erlang**) такое допускается.

» **Пропустили номер?** Узнайте на с. 104, как получить его прямо сейчас.

необходимо) и инвертирует ее. Инверсия нужна из-за того, что мы добавляем символы в начало результирующей строки в обратном порядке.

```
integer_to_rstring(Number, ExpectedLength) ->
  StringRepr = integer_to_list(Number),
  lists:reverse(StringRepr) ++ string:chars($0, ExpectedLength-length(StringRepr)).
```

Осталось проверить правильность работы нашего форматирования: вызов

```
format("DD++MM++(YYYY/YY)--hh--mm--ss--ms", #datetime{day = 1, month = 9, year = 2011, hour = 14, minutes = 1, seconds =0, milliseconds = 11})
```

возвращает нам ожидаемую результирующую строку

```
"01++09++(2011/11)--14--01--00--011".
```

Попробуем усложнить пример: создадим функцию для форматирования даты и времени в соответствии со строкой формата, причем значения спецификаторов для отдельных частей даты или времени должны задаваться пользователем нашей функции.

Для удобства работы с задаваемыми спецификаторами для отдельных частей даты объявим следующий тип записи (со значениями по умолчанию для спецификаторов, равными значениям из предыдущего примера):

```
-record(format_config, {day = "DD", month = "MM", short_year = "YY", year = "YYYY", hour = "hh", minutes = "mm", seconds = "ss", milliseconds = "ms"}).
```

Тут перед нами встает, пожалуй, главный в этом примере вопрос: можем ли мы использовать технику объявления нескольких вариантов функций, как в предыдущем примере.

Если бы ответ на этот вопрос был «да», то для нас все было достаточно просто. В действительности же объявить несколько вариантов функции у нас не получится: операция соответствия шаблону не позволяет указать (для параметров функции), что список начинается с другого списка произвольной длины, а в выражениях охраны нельзя использовать функции из модуля **lists** (в нашем случае – **lists:prefix/2**). Поэтому мы сами создадим некоторый аналог вариантов функции из предыдущего примера. Мы объявляем список обработчиков, которые будут возвращать либо результат обработки, либо атом **false**, означающий, что данный обработчик не может быть применен. После этого мы можем последовательно обработать строку формата, выбирая для очередного символа (или символов) подходящий обработчик из списка. Давайте посмотрим, как это все выглядит на практике. Начнем с главной функции – **smart_format/3**:

> «Некорректное объявление функции может принести сюрпризы.»

```
smart_format(FormatString, FormatConfig, DateTime) ->
  Handlers = [
  fun(Input, DT) -> format_part(Input, FormatConfig#format_config.day, DT#datetime.day) end,
  fun(Input, DT) -> format_part(Input, FormatConfig#format_config.month, DT#datetime.month) end,
  fun(Input, DT) -> format_year(Input, FormatConfig#format_config.year, DT#datetime.year) end,
  fun(Input, DT) -> format_syear(Input, FormatConfig#format_config.short_year, DT#datetime.year) end,
  fun(Input, DT) -> format_part(Input, FormatConfig#format_config.hour, DT#datetime.hour) end,
  fun(Input, DT) -> format_part(Input, FormatConfig#format_config.minutes, DT#datetime.minutes) end,
  fun(Input, DT) -> format_part(Input, FormatConfig#format_config.seconds, DT#datetime.seconds) end,
  fun(Input, DT) -> format_ms(Input, FormatConfig#format_config.milliseconds, DT#datetime.milliseconds) end,
  fun(Input, _DT) -> format_other_char(Input) end],
  process_format(FormatString, DateTime, Handlers, []).
```

В этом методе мы объявляем список обработчиков **Handlers** и вызываем функцию **process_format/4** для построения результирующей строки. Элементами списка обработчиков **Handlers** являются анонимные функции двух аргументов: остатка обрабатываемой строки формата и экземпляра записи типа **datetime** (которую мы определили выше). Следует отметить две вещи. Во-первых, обработчик для спецификатора года идет раньше обработчика для спецификатора короткого года, т.к. обычно они определяются разным количеством одних и тех же символов (как в предыдущем примере). Во-вторых, последним идет обработчик, который копирует первый символ остатка строки форматирования в результирующую строку (как в предыдущем примере).

Разберемся с функцией **process_format/4**. Она отвечает за обработку всей строки форматирования: из списка обработчиков выбирается один, который обрабатывает несколько символов начала остатка строки форматирования, после чего вызывается эта же функция рекурсивно (при помощи хвостовой рекурсии) с остатком от остатка строки форматирования и обновленной результирующей строкой. Ну и, конечно, есть вариант, обрабатывающий ситуацию, когда остаток строки форматирования равен пустой строке (когда мы обработали всю строку форматирования).

```
process_format([], _DateTime, _Handlers, Dest) ->
lists:reverse(Dest);
process_format(FormatString, DateTime, Handlers, Dest) ->
  {Data, FormatStringRest} = iterate_handlers(FormatString, DateTime, Handlers), process_format(FormatStringRest, DateTime, Handlers, Data ++ Dest).
```

Следующая задача, стоящая перед нами – это выбрать первый обработчик, который сможет обработать несколько символов начала остатка строки форматирования. Почему мы выбираем первый обработчик? Потому что у нас в конце списка обработчиков есть обработчик, который может обработать все символы (он просто копирует один символ из строки формата в результирующую строку). Если искать не первый возможный обработчик, а все, то мы можем получить неоднозначную ситуацию, когда остаток строки форматирования могут обработать более одного обработчика. Протокол взаимодействия с обработчиками у нас следующий: если обработчик возвращает кортеж, состоящий из атома **true**, обработанной порции данных и необработанного остатка строки формата, то этот обработчик является искомым; если же обработчик возвращает атом **false**, то необходимо искать обработчик дальше. Ну и естественно предусмотреть ситуацию, когда не найдено ни одного обработчика: мы сгенерируем исключение, т.к. такая ситуация является нарушением контракта. Все это реализовано в функции **iterate_handler/3**:

```
iterate_handlers(_Input, _DateTime, []) ->
erlang:error(bad_formatstring);
iterate_handlers(Input, DateTime, [Handler | OtherHandlers]) ->
  case Handler(Input, DateTime) of
    {true, Data, InputRest} -> {Data, InputRest};
    false -> iterate_handlers(Input, DateTime, OtherHandlers)
  end.
```

Теперь можно разобраться и с функциями, на основе которых строятся обработчики. Начнем с функции **format_part/3**, которая является общей функцией для построения обработчиков. В этой функции мы проверяем, начинается ли остаток строки формата с интересующего нас префикса. Если ответ положительный, то мы возвращаем кортеж, состоящий из атома **true**, преобразованного в инвертированную строку соответствующего компонента даты (который занимает два символа), и завершение остатка строки форматирования без префикса. Если ответ отрицательный, то мы возвращаем атом **false**. В общем, все в соответствии с протоколом взаимодействия между обработчиками и функцией **iterate_handler/3** (о которой мы говорили выше).

```
format_part(FormatString, FormatPart, DTPart) ->
  case lists:prefix(FormatPart, FormatString) of
    true -> {true, integer_to_rstring(DTPart, 2), string:sub_
string(FormatString, length(FormatPart)+1)};
    false -> false
  end.
```

Следующая функция, на основе которой мы строим обработчики – **format_year/3**. Она ничем особенно не отличается от функции **format_part/3**, за одним исключением: так как эта функция применяется для обработки спецификатора года, то она преобразовывает значение года в инвертированную строку из четырех символов.

```
format_year(FormatString, FormatYearPart, Year) ->
  case lists:prefix(FormatYearPart, FormatString) of
    true -> {true, integer_to_rstring(Year, 4), string:sub_
string(FormatString, length(FormatYearPart)+1)};
    false -> false
  end.
```

А вот чуть более интересная функция: она обрабатывает спецификатор короткого года (когда мы возвращаем две последних цифры года). В этом случае мы преобразовываем в инвертированную строку (в данном случае из двух символов) остаток от деления значения года на 100. В языке Erlang остаток от деления одного числа на другое вычисляется при помощи оператора **rem**.

```
format_syear(FormatString, FormatShortYearPart, Year) ->
  case lists:prefix(FormatShortYearPart, FormatString) of
    true -> {true, integer_to_rstring(Year rem 100, 2), string:sub_
string(FormatString, length(FormatShortYearPart)+1)};
    false -> false
  end.
```

Теперь давайте рассмотрим функцию **format_ms/3**. Эта функция примерно такая же, как и функции **format_part/3** и **format_year/3**: она применяется для обработки спецификатора миллисекунд и в процессе своей работы преобразовывает значение миллисекунд в инвертированную строку из трех символов.

```
format_ms(FormatString, FormatMillisecondsPart, Milliseconds)
->
  case lists:prefix(FormatMillisecondsPart, FormatString) of
    true -> {true, integer_to_rstring(Milliseconds, 3), string:sub_
string(FormatString, length(FormatMillisecondsPart)+1)};
    false -> false
  end.
```

И, наконец, у нас осталась последняя из функций, на основе которых мы строим обработчики остатка строки формата. Это функция **format_other_char/1**; она используется в том случае, когда другие обработчики не могут обработать несколько первых символов остатка строки формата. Все, что она делает – это проверяет, является ли первый элемент списка символом (при помощи выражения охраны), и если является, то добавляет этот символ в результирующую строку.

```
format_other_char([Char | Rest]) when is_integer(Char), Char > 0,
Char < 256 ->
  {true, [Char], Rest}.
```

Про функции, на основе которых строятся обработчики, можно добавить следующее: их все (кроме **format_other_char/1**) можно спокойно объединить в одну, принимающую, помимо остатка строки формата, спецификатора формата и соответствующего компонента даты, еще и длину строкового представления этого компонента даты. На практике мы бы так и поступили, но здесь для простоты восприятия материала автор решил этого объединения не делать.

Теперь осталось только проверить, что все у нас работает правильно. Пусть у нас дата и время определены так:

```
Time = #datetime{day = 7, month =12, year = 2011, hour = 23, min-
utes =32, seconds = 29, milliseconds = 7}.
```

Тогда вызов функции форматирования со значениями по умолчанию для спецификаторов формата

```
smart_format("DD=MM=YY+YYYY:hh:mm:ss#msfffff", #format_
config{}, Time)
```

вернет следующую результирующую строку:

```
"07=12=11+2011:23:32:29#007ffff".
```

Предположим, что мы определяем спецификаторы следующим образом (при помощи экземпляра записи **format_config**):

```
FormatConfig = #format_config{day = "day", month = "month",
year ="year", short_year = "syear", hour = "hour", minutes = "min",
seconds = "sec", milliseconds = "msec"}.
```

Тогда вызов функции форматирования для того же значения даты и времени

```
smart_format("day=month=syear+year:hour:min:sec#msecfffff",
FormatConfig, Time)
```

вернет такую же результирующую строку:

```
"07=12=11+2011:23:32:29#007ffff".
```

И напоследок, давайте поговорим, как собирать и запускать примеры. Для этого нужно свести весь приведенный код в модуль и экспортировать функции **format/2** и **smart_format/3** (добавив директивы **-module(datetime_formatter).** и **-export([format/2, smart_format/3]).** в начало модуля). Имя файла модуля должно совпадать с именем модуля без расширения (в нашем случае это **datetime_formatter.erl**). Для удобства работы, определения записей **datetime** и **format_config** лучше вынести в отдельный файл (например, в **datetime_formatter.hrl**) и подключать его везде, где требуется определение записи (например, так: **-include(datetime_formatter.hrl).**) Теперь мы можем собрать наш модуль и использовать его. Например, вызов функции **format/2** мы делаем следующим образом: **datetime_formatter:format(...)**.

Итак, мы увидели, что правильное объявление вариантов функции сильно облегчает реализацию функциональности и делает код более понятным, а неправильное объявление вариантов может привести к тому, что функция будет вести себя не так, как ожидалось. Ну и наконец, если мы не можем реализовать функциональность на вариантах функции, а задача предполагает, что должен происходить выбор одного из вариантов, то такой выбор достаточно легко реализовать на списке анонимных функций-обработчиков. А в следующей статье мы продолжим рассмотрение практических задач. LXF

Анонимные функции

Общее определение анонимной функции имеет следующий вид:

```
fun
(Pattern11,...,Pattern1N) [when GuardSeq1] -> Body1;
...;
(PatternK1,...,PatternKN) [when GuardSeqK] -> BodyK
end
```

Из этого определения видно, что анонимная функция может иметь несколько вариантов, как и обычная функция. И для нее действуют те же правила задания вариантов и выбора подходящего варианта.