



# Erlang: Тестируем!

Андрей Ушаков возобновляет серию о проверке правильности программ, написанных на языке Erlang.



Наш  
эксперт

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

В прошлый раз ([LXF182](#)) мы увидели, что процесс тестирования программных продуктов начинается еще с их разработки, когда программисты сразу же пишут тесты (модульные, функциональные, интеграционные и т. д.). Сегодня мы продолжим разговор и увидим, как выглядят тесты и как они пишутся.

Тестирование обычно проходит по следующему сценарию (независимо от того, ручное оно или автоматизированное): на «вход» подаются некоторые исходные данные, после чего на «выход» получается некоторый результат. После этого полученные результатирующие данные сравниваются с некоторыми заданными (эталонными) данными, хотя бывают такие тесты, в которых достаточно убедиться, что на «выходе» получены хоть какие-нибудь данные и не произошло прекращения работы из-за ошибки. Если полученный результат совпадает с эталоном, тест считается пройденным. Иначе считается, что мы тест не прошли; при автоматизированном тестировании об этом обычно уведомляют при помощи генерации некоторой специальной ошибки. Понимания этих принципов вполне достаточно, чтобы написать свою библиотеку для автоматизированного тестирования кода, но в большинстве случаев это будет изобретением велосипеда, т. к. для большинства платформ и языков предусмотрены общепринятые библиотеки (а то и несколько, как, например, библиотеки CppTest и GoogleTest для написания тестов на языке C++). Все эти библиотеки применяют следующий подход: в автоматизированных тестах расставлены специальные проверки. Если проверка выполнилась успешно, то ничего не происходит. Если же проверка не выполнилась, то генерируется специальная ошибка, после чего выполнение теста прекращается и тест считается не пройденным. Точно так же считается, что тест не пройден, если во время его выполнения сгенерируется какая-либо ошибка времени выполнения в исходном коде или в коде самого теста.

Давайте на простом примере рассмотрим, как выглядит подобный подход и подобные тесты. В качестве примера возьмем простой тест, написанный на языке C# с использованием библиотеки *NUnit* (операторы импорта соответствующих пространств имен мы здесь не приводим):

```
[TestFixture]  
public class SimpleTestSet  
{  
    [Test]  
    public void SimpleTest()  
    {  
        Assert.True(2 > 1);  
    }  
}
```

```
Assert.False(2 < 1);  
Assert.AreEqual(44, 40 + 4);  
Assert.Throws<InvalidOperationException>(() => {throw new  
InvalidOperationException();});  
Double value = 0;  
Assert.DoesNotThrow(() => value = Math.Cos(0));  
Assert.AreEqual(1.0, value, Double.Epsilon);  
}  
}
```

В этом примере класс **SimpleTestSet** определяет набор тестов; для этого он помечен атрибутом **[TestFixture]**. Обычно к набору тестов относят тестовые сценарии для проверки одной и той же функциональности (класса, подсистемы и т. п.). В нашем примере набор тестов состоит ровно из одного теста — метода **SimpleTest**; этот метод помечен атрибутом **[Test]**. Тест состоит из 6 выражений проверки: все эти проверки определяются в классе **Assert** из библиотеки *NUnit* (в соответствии с идеологией *xUnit* эти проверки называются «ассертами»). Выражение **Assert.True** служит для проверки переданного выражения на истинность. Выражение **Assert.False** проверяет переданное выражение на ложь. Выражение **Assert.AreEqual** проверяет равенство некоторого выражения эталонному значению (которое, конечно, также может быть выражением). Выражение **Assert.Throws** служит для проверки того, что во время выполнения некоторого выражения сгенерирована ошибка («отброшено исключение», в терминологии языка C#) определенного типа. Выражение **Assert.DoesNotThrow** служит для проверки того, что во время выполнения некоторого выражения не сгенерировано никакой ошибки. И, наконец, последнее выражение **Assert.AreEqual** позволяет проверить на равенство двух действительных числа с заданной точностью.

Может возникнуть вопрос: а для чего проверять возникновение ошибки при выполнении некоторого выражения? Ответ состоит в том, что генерация ошибки — это один из вариантов того, что может вернуться при вычислении выражения (помимо значения этого выражения, если оно есть). Иными словами, возникновение ошибок определенного типа при определенных условиях является контрактом данного выражения, точно таким же, как и возврат некоторого значения при заданных входных параметрах. И, очевидно, для более полного покрытия тестами нашего кода мы должны проверять и эти случаи.

А теперь перейдем непосредственно к языку Erlang и к написанию тестов на нем. Для этого воспользуемся библиотекой *EUnit*, доступной «из коробки» при установке среды выполнения Erlang. Первый шаг, который необходимо сделать при использовании библиотеки *EUnit* — это подключить ее. Делается это при помощи директивы **include\_lib**; она должна располагаться после определения модуля (директивы **module**), но до первого определения какой-либо функции. Вот пример использования директивы **include\_lib**:

```
-include_lib("eunit/include/eunit.hrl").
```

Подключение библиотеки *EUnit* (при помощи директивы **include\_lib**) приводит к следующим эффектам:

» Автоматически создается экспортруемая функция **test/O** в текущем модуле (из модуля, в котором мы использовали директиву **include\_lib**), и эту функцию можно использовать для запуска всех

## Установка EUnit из репозиториев

При работе со многими дистрибутивами операционной системы Linux появляется возможность устанавливать программное обеспечение из репозиториев пакетов Linux. Язык и среду выполнения Erlang тоже можно установить из таких

репозиториев. Однако при установке языка и среды выполнения Erlang следует обратить внимание на следующее: в некоторых версиях Linux (например, в Linux Mint 16) библиотека *EUnit* устанавливается как отдельный пакет.

## Значения выражений в макросах

В языке Erlang все является выражением с некоторым значением. Это же справедливо и касательно макросов из библиотеки *EUnit*. Большинство макросов предназначены либо просто для некоторой проверки, либо для вывода информации на консоль. К таким макросам относятся следующие: **assert**, **assertNot**, **assertEqual**, **assertNotEqual**, **assertMatch**, **assertNotMatch**, **assertError**, **assertExit**, **assertThrow**,

**assertException**, **assertNotException**, **assertCmd**, **assertCmdStatus**, **assertCmdOutput**, **debugHere**, **debugMsg** и **debugFmt**. Возвращаемое значение таких макросов обычно не важно, но тем не менее полезно будет знать, что эти макросы возвращают атом **ok** (либо генерируют ошибку, когда в макросах проверки проверка не проходит). Макрос **cmd** выполняет команду, переданную ему в качестве

аргумента, и возвращает вывод этой команды в виде одной строки (при этом вывод нормализуется: символом разрыва строки становится одиночный символ LF). Макросы **debugVal** и **debugTime** вычисляют выражение (переданное в качестве аргумента), выводят некоторую информацию на консоль, связанную с этим выражением, и возвращают значение выражения.

тестов, определенных в текущем модуле. Функция **test/0** не будет автоматически создана в текущем модуле, если такая функция в нем уже есть или если запуск тестов отключен.

» Все функции вида **XXX\_test/0** и **XXX\_test/\_0** (здесь **XXX** — любое разрешенное имя функции) автоматически экспортируются из текущего модуля (из модуля, в котором мы использовали директиву **include\_lib**). Автоматического экспорта подобных функций не будет, если отключен запуск тестов и/или автоматический экспорт.

» Можно использовать ряд макросов для написания тестов (о них мы поговорим ниже).

Следующий шаг, который следует сделать при использовании библиотеки *EUnit* — это определить методы, содержащие интересующие нас тесты. Имена таких методов с тестами должны иметь следующий вид: **XXX\_test/0**, где **XXX** — любое разрешенное имя функции (в данном случае выступающее как префикс). Если во время выполнения такого метода не генерируется ошибки, то считается, что тест выполнился успешно. В противном случае считается, что выполнение теста провалилось. Если тест не закончит свое выполнение в течение некоторого времени (например, застикнется), будет считаться, что выполнение теста также провалилось. Как уже говорилось выше, внутри теста мы делаем некоторые действия и обычно проверяем полученные результаты. Если результаты не совпадают с эталонными данными, то в таком случае можно генерировать ошибку. Можно также воспользоваться операцией соответствия шаблону; точнее, тем, что полученных данные не соответствуют шаблону — в этом случае ошибка генерируется автоматически. Но библиотека *EUnit* предлагает более понятную альтернативу подобным инструкциям: макросы **assertXXX** (эти макросы напоминают класс **Assert** и его методы из библиотеки *NUnit*, которые мы показали выше). Принцип работы этих макросов следующий: они проверяют некоторое условие (зависящее от вида макроса), и если результат проверки условия будет ложным, генерируется некое исключение.

Макросы **assertXXX** являются «сердцем» тестов, написанных при помощи библиотеки *EUnit*: практически в каждом таком teste будет, по крайней мере, один из макросов **assertXXX**. Так что с этими макросами необходимо познакомиться поближе. Простейшим макросом является **assert(BoolExpr)**: он проверяет выражение **BoolExpr** на истинность. Макрос **assertEqual(Expect, Expr)** сравнивает значение выражения **Expr** с ожидаемым значением **Expect**. Макрос **assertMatch(GuardedPattern, Expr)** сравнивает значение выражения **Expr** с заданным шаблоном **GuardedPattern**, который может содержать выражение охраны (отделяемое от тела шаблона при помощи ключевого слова **when**) как с одной, так и с несколькими проверками. Когда проверок в выражении охраны несколько, только символ ‘,’ будет допустимым разделителем между ними. Макросы **assertError(TermPattern)**,

**Expr**), **assertExit(TermPattern, Expr)** и **assertThrow(TermPattern, Expr)** проверяют, что при вычислении выражения **Expr** сгенерирована ошибка, соответствующая шаблону **TermPattern**. Макрос **assertError** ожидает, что ошибка будет класса **error** (т.е. сгенерирована функциями **error/1,2**). Макрос **assertExit** ожидает, что ошибка будет класса **exit** (т.е. сгенерированная функциями **exit/1,2**). Наконец, макрос **assertThrow** ожидает, что ошибка будет класса **throw** (т.е. сгенерирована функцией **throw/1**). Вместо этих трех макросов можно использовать более общий макрос **assertException(ClassPattern, TermPattern, Expr)**. Он ожидает, что при вычислении выражения **Expr** сгенерируется ошибка класса **ClassPattern** (**error**, **exit** или **throw**), соответствующая шаблону **TermPattern**. Для макросов **assert**, **assertEqual**, **assertMatch** и **assertException** определены парные макросы **assertNot**, **assertNotEqual**, **assertNotMatch** и **assertNotException** соответственно — для них проверка условия будет успешна в том случае, когда результат проверки условия для исходного макроса будет ложным. Для макросов **assertError**, **assertExit** и **assertThrow** парных макросов нет. Это все макросы, которые нам доступны для проверки результата вычисления некоторого выражения (мы помним, что в языке Erlang все является выражением).

По поводу этих макросов отметим следующее. Во-первых, макросы **assertNotEqual**, **assertNotException** и **assertNotMatch** не документированы; об их наличии можно узнать, посмотрев исходный код библиотеки *EUnit* (из файла **eunit.hrl**). Так что по поводу этих макросов может возникнуть вполне логичный вопрос: стоит ли их использовать? Однозначный ответ, конечно, тут дать невозможно, но можно посоветовать следующее: наибольшая опасность, которая может возникнуть при использовании этих макросов — что авторы библиотеки *EUnit* как-то изменят их интерфейс или удалят. Но если посмотреть на эти макросы внимательно, станет понятно, что вероятность каких-либо изменений в них мала. Во-вторых, макрос **assertNotException** проверяет, что во время вычисления выражения не сгенерирована ошибка определенного класса, соответствующая некому шаблону. Проверить, что во время вычисления выражения не сгенерировано никакой ошибки, несколько нетривиально; следующий пример показывает, как можно осуществить подобную проверку:

```
?assertNotException(_, _, error(some_error_reason)).
```

К сожалению, при использовании данного подхода есть одна проблема: компилятор выдает предупреждение на данную проверку (и это предупреждение никак не обойти). Если использовать опцию трактовки предупреждений как ошибок при компиляции, то тест с данной проверкой нельзя будет скомпилировать. Чтобы обойти эту проблему, можно самому написать макрос, более тривиальный в использовании: он будет проверять, что вычисление выражения прошло без ошибок (без генерации каких-либо ошибок), и не будет вызывать предупреждений компилятора. Вот один из вариантов такого макроса:

»

» Не хотите пропустить номер? Подпишитесь на [www.linuxformat.ru/subscribe/!](http://www.linuxformat.ru/subscribe/)



## Тесты без макросов assertXXX

Макросы `assertXXX` предназначены для проверки результата вычисления выражений по некоторым исходным данным (здесь под результатом вычисления выражения мы понимаем как значение этого выражения, так и генерацию ошибки при вычислении). Если результат вычисления выражения не пройдет проверку, сгенерируется специальная ошибка, сигнализирующая библиотеке *EUnit* о том, что данный тест провалился. Если в результате

вычисления какого-либо выражения будет сгенерирована некоторая произвольная ошибка (никак не связанная с проверкой), то для библиотеки *EUnit* это также будет означать, что выполнение теста провалилось. Поэтому, если цель теста — просто проверить, что при вычислении некоторых выражений не происходит генерации каких-либо ошибок, вполне вероятно, что такой тест не будет вообще содержать макросы вида `assertXXX`.

Бывает ситуации, когда мы хотим просто проверить, что при вычислении некоторого выражения не сгенерировано какой-либо ошибки. Можно, конечно, в teste просто вычислить значение этого выражения, но при таком подходе явно не видно наше желание сделать подобную проверку. Другими словами, в некоторых ситуациях необходимо для наглядности явно указать, что при вычислении выражения мы не ждем никаких исключений.

```
-define(assertDoesNotException(Expr), ?assertEqual(true, try  
(_=Expr), true catch _:_ -> false end)).
```

И, наконец, в библиотеке *EUnit* отсутствуют макросы для сравнения действительных чисел с произвольной точностью (такое сравнение нужно из-за ошибок вычислительной математики).

Помимо приведенных выше макросов для проверки результатов вычисления выражений, в библиотеке *EUnit* есть еще ряд макросов для запуска на выполнения некоторой внешней команды и проверки результатов ее выполнения. Макрос `assertCmd(CommandString)` запускает на выполнение внешнюю команду, задаваемую строкой `CommandString`, ожидает окончания ее работы и проверяет значение кода возврата. Если значение кода возврата будет 0 (то есть команда успешно выполнилась), то проверка пройдет успешно; в противном случае проверка не пройдет. Макрос `assertCmdStatus(N, CommandString)` делает то же, что и макрос `assertCmd`, за одним исключением: код возврата выполнения команды сравнивается не с 0, а с некоторым заданным значением `N`. Макрос `assertCmdOutput(Text, CommandString)` запускает на выполнение внешнюю команду, задаваемую строкой `CommandString`, ожидает окончания ее работы и сравнивает вывод, сгенерированный командой (в виде единой строки), с ожидаемым выводом `Text`. Перед сравнением вывод от команды нормализуется: символом разрыва строки становится одиночный символ LF. И, наконец, макрос `cmd(CommandString)` запускает на выполнение внешнюю команду, задаваемую строкой `CommandString`, ожидает окончания ее работы и сравнивает значение кода возврата с 0. Главное отличие этого макроса от остальных (в особенности от `assertCmd`) заключается в том, что макрос `cmd` в случае успешной проверки кода возврата возвращает весь вывод, сгенерированный командой, в виде единой строки. При этом вывод, сгенерированный командой, нормализуется: символом разрыва строки становится одиночный символ LF. Очевидно, что значение, возвращаемое макросом `cmd`, можно использовать далее в тестах.

Есть еще ряд макросов, использование которых упрощает написание тестов — в особенности их отладку. Это макросы для вывода отладочной информации на консоль. Макрос `debugHere` выводит на консоль текущий файл и номер текущей строки. Макрос `debugMsg(Text)` выводит значение выражения `Text` на консоль. При этом `Text` может быть обычной строкой, атомом или IO-списком. Макрос `debugFmtFmtString, Args)` форматирует строку при помощи функции `io_lib:format/2` и выводит результат на консоль. Макрос `debugVal(Expr)` выводит как исходный код выражения `Expr`, так и его значение, а также возвращает значение выражения. Так, например, выражение `?debugVal(f(X))` выведет строку “`f(X) = 666`” и вернет 666 (при условии, что значение выражения `f(X)` равно 666 при данном значении аргумента `X`). И, наконец, макрос `debugTime(Text, Expr)` выводит комбинацию текста `Text`

и времени, необходимого для вычисления значения выражения `Expr`; этот макрос возвращает значение выражения `Expr`. Так, например, выражение `?debugTime(`some long calculation`, f(X))` выведет строку “`some long calculation: 1.002 s`” (если для вычисления выражения `f(X)` понадобится 1,002 секунды) и вернет значение выражения `f(X)`.

Но хватит слов: посмотрим, как мы будем писать тесты и использовать описанные выше макросы на практике.

```
-module(eunit_example_tests).  
-include_lib("eunit/include/eunit.hrl").  
example1_test() ->  
    ?assert(4 == 2*2),  
    ?assertNot(5 == 2*2),  
    ?assertEqual(4, 2*2),  
    ?assertNotEqual(5, 2*2),  
    ?assertMatch([1 | _Rest], [1, 2, 3, 4]),  
    ?assertMatch([1 | Rest] when length(Rest) == 2;length(Rest) ==  
3, [1, 2, 3, 4]),  
    ?assertNotMatch([2, _Rest], [1, 2, 3, 4]),  
    ?assertNotMatch([1 | Rest] when length(Rest) == 1;length(Rest)  
== 2, [1, 2, 3, 4]),  
    ?assertError(some_error_reason, error(some_error_reason)),  
    ?assertExit(some_exit_reason, exit(some_exit_reason)),  
    ?assertThrow(some_throw_reason,  
throw(some_throw_reason)),  
    ?assertException(error, some_error_reason,  
error(some_error_reason)),  
    ?assertException(exit, some_exit_reason,  
exit(some_exit_reason)),  
    ?assertException(throw, some_throw_reason,  
throw(some_throw_reason)),  
    ?assertNotException(error, _, 2+3),  
    ?assertNotException(exit, some_error_reason,  
error(some_error_reason)),  
    ?assertNotException(error, some_exit_reason,  
error(some_error_reason)),  
    ?assertNotException(error, some_error_reason),  
    ?assertCmd("echo hello world !"),  
    ?assertCmdStatus(127, "some-unknown-command"),  
    ?assertCmdOutput("hello world !\n", "echo hello world !"),  
    ?assert(?cmd("echo hello world !") /=""),  
    ?debugHere,  
    ?debugMsg("Some debug message"),  
    ?debugFmt("Some format message: ~p", [{format_data, 666}]),  
    ?assertEqual(4, ?debugVal(2*2)),  
    ?assertEqual(ok, ?debugTime("some long calculating",  
timer:sleep(1000))).
```

В данном примере мы объявляем модуль `eunit_example_tests`, подключаем библиотеку *EUnit* (директивой `include_lib`)

» **Пропустили номер?** Узнайте на с. 108, как получить его прямо сейчас.

и объявляем функцию `example1_test/0`, которая и будет содержать наш тест. Функция `example1_test/0` имеет вид `XXX_test/0`, так что, как говорилось выше, функция `example1_test/0` экспортируется из модуля `eunit_example_tests` автоматически; поэтому модуль `eunit_example_tests` не содержит директивы `export`.

Рассмотрим функцию `eunit_example_tests:example1_test/0` более пристально: она содержит наш тест, в котором мы используем все вышеописанные макросы (а некоторые и не один раз). Использование макросов `assert`, `assertNot`, `assertEqual` и `assertNotEqual` тривиально и не должно вызывать вопросов (не забудем, что имя макроса всегда начинается с символа '?'). По поводу макросов `assertEqual` и `assertNotEqual` следует заметить следующее: ожидаемое значение ( некоторое предопределенное значение, с которым сравнивается значение выражения) всегда должно быть левым аргументом, хотя с точки зрения компилятора это не важно (но важно с точки зрения семантики этих макросов). С макросами `assertMatch` и `assertNotMatch` все несколько интереснее: шаблон, с которым мы сопоставляем значение выражения, может содержать выражение охраны (идущее после ключевого слова `when`). При этом выражение охраны может состоять из нескольких проверок, но для их разделения разрешено использовать только символ ';' (это означает, что выражение охраны истинно, если хотя бы одна из проверок в нем истинна). Если мы используем выражения охраны, например, для определения разных вариантов функций, то для разделения проверок в таких выражениях охраны можно использовать также символ ';' (означающий, что выражение охраны истинно, если все проверки в нем истинны). Однако в макросах мы такой разделитель между проверками использовать не можем, поскольку этот разделитель будет неотличим от разделителя аргументов в самом макросе (которым также является символ ';'). В нашем примере мы используем и простые шаблоны без выражения охраны, и шаблоны с выражением охраны с несколькими проверками. Использование макросов `assertError`, `assertExit`, `assertThrow` и `assertException` также достаточно тривиально: мы ожидаем, что в результате вычисления интересующего нас выражения сгенерируется ошибка определенного класса, соответствующая шаблону. Класс ошибки определяется методом, при помощи которого ошибка была сгенерирована, а сама ошибка определяется аргументом, передаваемым в метод для генерации ошибки. В нашем примере мы просто генерируем определенную ошибку определенного класса (в качестве выражения, значение которого необходимо вычислить) для проверки работы макросов `assertError`, `assertExit`, `assertThrow` и `assertException`. А вот использование макроса `assertNotException` несколько нетривиально: в нем мы проверяем, что не сгенерирована ошибка определенного класса, соответствующая шаблону. Этой проверке удовлетворяют следующие три случая, когда во время вычисления выражения

- ❶ никакой ошибки не сгенерировано;
- ❷ сгенерирована ошибка другого класса;
- ❸ сгенерирована ошибка ожидаемого класса, не соответствующая заданному шаблону.

Все эти три случая мы показали в нашем примере.

Макросы, предназначенные для взаимодействия со внешним миром — `assertCmd`, `assertCmdStatus`, `assertCmdOutput` и `cmd` — сами по себе также достаточно тривиальны; нетривиальными могут быть команды, используемые в данных макросах. Но в нашем примере мы показываем, как использовать именно макросы. Поэтому в этих макросах мы используем очень простые команды (а точнее, ограничиваемся командой `echo`), которые всегда будут выполняться успешно. Исключение составляет только пример с использованием макроса `assertCmdStatus`: здесь нам нужно получить код возврата выполнения команды, отличный от 0. Для этого мы пользуемся следующим фактом: код возврата выполнения неизвестной команды в операционных

системах семейства \*nix всегда равен 127 (один из стандартных кодов возврата). В примере для макроса `assertCmdStatus` мы пытаемся выполнить несуществующую команду и сравниваем код возврата со значением 127.

И, наконец, в завершение нашего теста мы демонстрируем использование отладочных макросов `debugHere`, `debugMsg`, `debugFmt`, `debugVal` и `debugTime`. Причем макросы `debugHere`, `debugMsg` и `debugFmt` просто выводят информацию на экран, а макросы `debugVal` и `debugTime` еще и вычисляют значение некоторого выражения и возвращают вычисленное значение. Поэтому в примерах с макросами `debugVal` и `debugTime` мы проверяем значение, которое они возвращают. Макрос `debugTime` предназначен в первую очередь для измерения времени вычисления некоторого выражения; в нашем примере для простоты в качестве такого выражения мы используем паузу в 1000 миллисекунд. При этом измеренное время, необходимое для вычисления этого выражения, будет всегда чуть больше этой паузы (и, что естественно, каждый раз разное). В результате выполнения этого теста мы получим на консоли нечто подобное:

```
eunit_example_tests.erl:27:<0.39.0>: <
eunit_example_tests.erl:28:<0.39.0>: Some debug message
eunit_example_tests.erl:29:<0.39.0>: Some format message:
{format_data,666}
eunit_example_tests.erl:30:<0.39.0>: 2 * 2 = 4
eunit_example_tests.erl:31:<0.39.0>: some long calculating: 1.002 s
Test passed.
```

Тесты мы писать научились (хотя пока и самые простые) — теперь на повестке дня следующий вопрос: как их запускать. Для запуска тестов на выполнение достаточно использовать одну из функций `eunit:test/1,2`. В простейшем случае, функция `eunit:test/1` принимает имя модуля, находит все тесты в этом модуле и выполняет их. Однако и тут есть масса нюансов. Пусть, например, имя модуля, которое мы передаем в функцию `eunit:test/1`, будет `m`. Тогда библиотека `EUnit` будет искать тесты как в модуле `m`, так и в модуле `m_tests`, если такой модуль существует. Если модуля `m` не существует, но существует модуль `m_tests` и мы передадим в функцию `eunit:test/1` в качестве имени модуля `m`, то получим ошибку времени выполнения. И, наконец, если существуют модули `m`, `m_tests` и `m_tests_tests`, и мы передадим в функцию `eunit:test/1` в качестве имени модуля `m_tests`, то библиотека `EUnit` будет искать тесты только в модуле `m_tests`. В более сложных случаях, в функцию `eunit:test/1` можно передавать и другие объекты, например, список имен модулей с тестами. Но об этом мы поговорим в другой раз. Функция `eunit:test/2` позволяет задать список опций в качестве второго аргумента. На данный момент в качестве такой опции можно использовать только атом `verbose`. Использование данной опции приводит к тому, что библиотека `EUnit` выводит некоторую дополнительную информацию на консоль.

Сегодня мы только начали разбираться в том, какие средства для написания тестов есть в библиотеке `EUnit`. Также, мы написали специальный (синтетический) тест, демонстрирующий использование всех макросов. В следующий раз мы продолжим обсуждать возможности, доступные в библиотеке `EUnit`. [\[xf\]](#)

## Что такое IO-список

В документации по языку Erlang и его библиотекам можно найти следующее определение IO-списка: `iolist = maybe_improper_list(byte() | binary() | iolist(), binary() | [])`. Из этого определения видно: IO-список — это список, элементами которого могут быть байты, байтовые строки и другие IO-списки. Вот такое рекурсивное определение...