STEAD3: Программирование модулей

Петр Косых

22 апреля 2018 г.

Оглавление

Глава 1

Модули STEAD3

Данное руководство описывает основы разработки модулей STEAD3. Поэтому в дальнейшем изложении слово “модуль” означает именно модуль написанный в рамках STEAD3.

Что такое модуль? Часто ошибочно считают, что модуль – это любой отдельный файл игры, содержащий служебные функции, структуры и объекты. На самом деле это не так. Если вам необходимо выделить служебный код в отдельные игровые файлы так, чтобы он выполнялся в момент старта игры – вам достаточно включить такой файл инструкцией include:

```
include ”mylib”
```

Модули stead3 пишутся следуя специальным соглашениям, при нарушении которых вы получите нерабочую игру. Данные соглашения и описываются в этом руководстве. Очень часто в оформлении своего кода в виде модуля просто нет никакой необходимости. Написание модуля целесообразно в тех случаях, когда требуемая функциональность может быть реализована за счет тесного взаимодействия с движком STEAD3 и представляет ценность не только в рамках вашей игры. Примерами модулей могут быть:

- модуль форматирования вывода;
- модуль нестандартного управления игрой;
- модуль генерации изображений нотного стана…

И так далее.

Глава 2

Игровой мир

Важным отличием модуля от просто библиотечного файла является время жизни модуля. Игровой мир (объект game, игрок, все объекты и переменные игры) создается в момент запуска игры. При подгрузке частей игры с помощью gamefile, а также при загрузке сохранений игры – игровой мир уничтожается, чтобы затем снова создаться заново. В момент уничтожения мира, все созданные объекты и переменные перестают существовать.

Когда вы включаете код с помощью include, весь этот код будет выполнятся в момент каждого создания мира. Обычно, это подразумевается автором как само собой разумеющееся поведение. Но в случае модуля, это не так!

Модуль загружается один раз, во время создания игровой сессии и существует все то время, пока выполняется игровой код. То есть, после уничтожении игрового мира и при его новом создании, уже загруженные ранее модули не загружаются повторно.

Это означает, например, что в модуле *нельзя* определять глобальные переменные, объекты и комнаты так, как это делается в обычном коде игры. Например, модуль не может содержать строки вида:

```
-- мой неправильный модуль
global 'MYVAR' (10);
room { nam = 'комната'; };
```

Дело в том, что и MYVAR и комната будут уничтожены при первой же загрузки игры из файла сохранения, так как игровой мир уничтожается целиком, а переменная MYVAR и комната – часть этого игрового мира.

Если подумать, это становится понятным с той точки зрения, что модуль не должен в общем случае создавать какие-то объекты и глобальные переменные

в игровом мире игры. Ведь модуль может работать с любой игрой, и не должен учитывать особенности игрового мира конкретной игры.

Создание глобальных функций тоже не является хорошим тоном при написании модулей. Если вам нужна служебная функция в рамках самого модуля, объявляйте ее с local (то же касается и переменных):

```
local A = 10
local function myfunc(a, b)
    return (a * b) / A
end
```

Тогда такая функция будет доступна в рамках модуля.

Глава 3

Системный объект как интерфейс

После прочтения прошлой главы мог возникнуть резонный вопрос. Если все переменные и объекты модуля уничтожаются – то как вообще взаимодействовать с модулем? Если модуль должен предоставлять какой либо интерфейс, а также сохранять свое состояние, то в модуле следует создать системный объект (чье имя начинается с символа ‘@’). Подробнее про системные объекты можно прочитать в документации STEAD3. Важно то, что такие объекты *не уничтожаются* в процессе смерти мира, и таким образом играют роль интерфейса к модулю. Например:

```
-- модуль testmod.lua
obj {
    nam = '@testmod';
    hello = function(s)
        dprint("Hello world");
    end;
}
```

Такой модуль уже можно использовать:

```
loadmod 'testmod'
_'@testmod'.hello() -- получили объект '@testmod' и вызвали метод hello()
```

Но это не слишком удобно. Следующий вариант модуля:

```
-- модуль mymodule.lua
local mod = obj { -- локальная ссылка на объект
    nam = '@testmod';
}

function mod.hello(a) -- вариант определения метода вне объекта
    dprint("Hello world")
end

testmod = mod -- глобальный объект testmod -- интерфейс к модулю
```

Использование модуля:

```
loadmod 'testmod'
testmod.hello()
```

Сделаем вариант модуля с переменной, которая должна сохранятся:

```
-- модуль mymodule.lua
local mod = obj { -- локальная ссылка на объект
    nam = '@testmod';
    num = 0; -- переменная
}

function mod.hello(a)
    dprint("Hello world", mod.num)
    mod.num = mod.num + 1
end

testmod = mod -- глобальный объект testmod -- интерфейс к модулю
```

Итак, подводя итоги:

- интерфейсом к модулю служит системный объект, имя которого совпадает с именем модуля;
- все переменные, которые нужно сохранять, должны быть переменными этого объекта;
- вы можете вызывать методы через модуль.метод().

Глава 4

События

Итак, модуль существует все время жизни игровой сессии, во время которой игровой мир может неоднократно уничтожаться и создаваться снова. Допустим, автору модуля необходимо выполнять какие то действия в момент создания игрового мира (например, помещать что то в инвентарь игрока). Как это сделать? Модуль может регистрировать специальные функции-обработчики некоторых событий. Например, для отслеживания события "создание игрового мира", мы регистрируем обработчик init:

```
stead.mod_init(function()
        declare {
                game = std.ref 'game',
                pl = std.ref 'player',
        }
end)
```

stead – это интерфейсный объект для доступа к низкоуровневым функциям INSTEAD. Часто, удобно сделать локальную ссылку std, и пользоваться ей, например:

```
local std = stead -- сделали ссылку на stead

std.mod_init(function()
        declare {
                game = std.ref 'game',
                pl = std.ref 'player',
        }
```

```
end)
```

В дальнейшем изложении всегда будет использоваться std вместо stead.

Приведенный код – инициализация стандартной библиотеки stdlib из STEAD3. Как видим, тут происходит объявление переменных-ссылок game и pl для удобства работы с этими объектами.

Итак, регистрация обработчика выполняется с помощью вызова:

```
std.mod_тип_события(функция, [приоритет])
```

Параметр [приоритет] (чем меньше – тем приоритетней) является необязательным и определяет порядок вызова вашей функции относительно других функций для данного события, например:

```
local function init()
      dprint(”Очень ранний init”);
end
std.mod_init(init, -10);
```

Перечислим все события, для которых можно зарегистрировать свой обработчик.

- init - инициализация мира. Вызывается перед init() игры;
- done - деинициализация мира;
- start - запуск игры после инициализации мира (непосредственно перед вызовом start() игры). В обработчик придет параметр load (true – если это загрузка из файла сохранения);
- cmd - вызывается перед выполнением команды. Если вернет не nil – воспринимается движком как реакция на команду. В качестве параметра
- массив (команда, параметр1, параметр2, …);
- step - вызывается после выполнения команды. В качестве параметра – true (команда выполнена), или false (ошибочная команда); Выполняется после game:step();
- save - вызывается после сохранения динамических объектов игры в файл сохранения. В качестве параметра – указатель на файл сохранения (Lua).

Глава 5

Классы

Все объекты в INSTEAD, кроме переменных, атрибутов и обработчиков содержат методы (функции), которые вызываются при работе движка.

Когда в коде вашей игры, вы пишете что то вроде:

```
obj {
    nam = 'стол';
    dsc = "Тут стоит {стол}.";
}
```

То вы создаете таблицу Lua, в которой кроме nam и dsc определены многие другие вещи. Например, метод disable(). Когда вы пишете:

```
obj {
    nam = 'стол';
    dsc = "Тут стоит {стол}.";
}:disable()
```

Вы у созданного только что объекта 'стол' вызываете метод disable(). (Вызывается функция disable объекта и в качестве 1-го параметра передается этот же объект).

На самом деле, даже когда вы создаете объект с помощью obj, вы неявно вызываете метод (функцию) new у класса obj: obj.new { описатель объекта }

Классы, удобный способ описать иерархию объектов INSTEAD, а так же реализовать свои типы объектов. Например, вот как определяется класс obj (с сокращениями):

```
std.obj = std.class { -- определяем класс
    __obj_type = true; -- тип объектов (функция is_obj)
    with = function(self, ...)
         -- ... реализация конструкции :with
    end;
    new = function(self, v)
        -- реализация конструктора obj { }
    end;
    -- другие методы
    -- ...
    display = function(self) -- отображение объекта
        local d = std.call(self, 'dsc')
        return d
    end;
    -- другие методы
    -- ...
};
```

Обратите внимание на метод display, этот метод служит для отображения объекта в сцене. Как видите, он просто вызывает 'dsc' у объекта, с помощью std.call (вызов функции или строки).

Теперь представим себе, что вы хотите сделать класс объектов, у которых в темноте пропадает описатель. Признаком темноты является переменная darkroom в текущей комнате, тогда:

```
darkobj = std.class ({ -- определяем класс
    __darkobj_type = true; -- тип объектов (функция is_obj)
    display = function(self) -- отображение объекта
        if not here().darkroom then
            local d = std.call(self, 'dsc')
            return d
        end
    end;
}, std.obj);


darkobj {
```

```
    nam = 'кот';
    dsc = '{Кот} виден при свете.'
}
```

Обратите внимение на скобки () и std.obj в последней строке. Таким образом мы дали понять, что наследуемся от класса std.obj. Все методы объекта, созданного при помощи darkobj, будут такими же как и у obj, кроме метода display().

Если говорить об иерархии объектов STEAD3, то:

- room пронаследован от obj;
- dlg пронаследован от room;
- std.phr (фразы) пронаследован от obj;

std.dlg, std.room, std.phr и std.obj это внутренние имена классов. В стандартной библиотеке stdlib, которая включается в вашу игру, определены такие переменные;

```
local std = stead
local type = std.type
std.rawset(_G, 'std', stead) -- определим std для всех
include = std.include
loadmod = std.loadmod
rnd = std.rnd
rnd_seed = std.rnd_seed
p = std.p
pr = std.pr
pn = std.pn
pf = std.pf
obj = std.obj
stat = std.stat
room = std.room
menu = std.menu
dlg = std.dlg
me = std.me
here = std.here
from = std.from
```

```
new = std.new
delete = std.delete
gamefile = std.gamefile
player = std.player
dprint = std.dprint
```

Это сделано для удобства автора (легче написать include, вместо std.include или stead.include). Если же вы разрабатываете свой *модуль*, то рекомендуется пользоваться внутренними именами с префиксом std. Причем std определять в начале модуля:

```
local std = stead
```

Это обезопасит ваш модуль от коллизий с игровыми глобальными переменными.

Глава 6

Цикл обработки команд

Жизненный цикл игры на INSTEAD это обработка команд. Примерно как сервер отвечает на запросы клиентов, INSTEAD получает команды и отвечает на них текстом. Команды формируются интерпретатором в зависимости от того, какие действия выполняет игрок. Команда – это слово, за которым может быть список аргументов.

Примеры команд STEAD3:

- load <файл>; - загрузка
- save <файл>; - выгрузка
- use <объект>,<объект>; - использование объекта на объект
- use <объект>; - использование объекта в инвентаре
- act <объект>; - действие на объект
- act <$объект>,аргументы…; - ссылка на системный объект
- go <переход>; - переход
- look; - осмотреться
- inv; - получить инвентарь
- way; - получить список переходов

В качестве объектов используются идентификаторы в виде цифр, которые движок сам сопоставляет каждому видимому на сцене объекту.

Каким образом STEAD3 обрабатывает команды?

Интерпретатор работает с Lua частью через специальный системный объект iface (переменная ссылка на "@iface").

Этот объект создается внутри STEAD3, а для графической версии интерпретатора INSTEAD, пересоздается специальным модулем ext/gui.lua. Выполнение команды это вызов iface:cmd(команда).Обратите внимание на ':'. Такая запись это синоним: iface.cmd(iface, команда).

Команда разбивается на части, и записывается в специальный массив std.cmd. Так, в случае команды load, в std.cmd[1] будет записано "load", а в std.cmd[2] – путь к файлу.

Затем происходит вызов game:cmd(std.cmd). Это и есть то место, где команда обрабатывается игрой.

После чего вывод (возвращаемое значение) пропускается через iface:fmt(). Который в свою очередь вызывает std.fmt.

В std.fmt ссылки в выводе игры {} превращаются в ссылки для интерпретатора.

Глава 7

Примеры модулей

7.1 Пишем компас

В качестве примера разберем модуль, реализующий простой компас. Наш компас будет представлен ссылками С/Ю/В/З в инвентаре. При нажатии на одну из ссылок, будет осуществляться переход по атрибуту комнаты: to_n, to_s, to_e, to_w соответственно.

Начнем с того, что создадим файл compass.lua, в котором создадим объект @compass:

```
local std = stead

obj {
    nam = '@compass';
}
```

Наш компас – это фактически набор ссылок: С, Ю, В, З. При нажатии на ссылку, мы должны выполнить walk. Как отобразить ссылки в инвентаре?

Мы можем поступить двумя способами:

1. создать для каждого направления свой объект;
2. воспользоваться подстановками.

Первый способ плох тем, что не позволяет нам произвольно форматировать вывод (а мы хотим, чтобы компас был похож на компас), поэтому подстановки, в данном случае, будут удобнее.

```
obj {
    nam = '@compass';
    disp = false; -- не показывать объект @compass
}:with {
    obj {
        nam = '$compass';
        act = function(s, w)
            std.p(fmt.c ("{@compass n| C}\n{@compass w|З} {@compass e|В}\n{@compas
        end;
    }
}
```

В объект @compass (который стал невидимым, см disp = false), мы вложили объект $compass. Как вы помните, это подстановки. Движок, встретив ссылки вида {$compass} (а именно такая ссылка будет в инвентаре, если мы поместим туда объект компас), вызовет act у объекта $compass и его результат отобразит в выводе. Мы и воспользовались этим! Функция act у $compass формирует строку ссылок: С, З, В, Ю. Причем в качестве объекта реакции выступает @compass. Это значит, что добавив обработчик inv в @compass мы получим вызов этого метода как реакцию на нажатие ссылки:

```
obj {
    nam = '@compass';
    disp = false;
    inv = function(s)
         p [[Вы нажали на ссылку.]]
    end;
}:with { ...
```

Но как определить, в какую сторону мы хотим идти? Ссылки нашего компаса имеют вид:

```
{@compass n|C}
```

Это значит, что движок, при нажатии на такую ссылку, сформирует команду:

```
use @compass,n
```

Которая и приведет к вызову метода inv у объекта @compass. Но как получить второй параметр, в данном случае n? К сожалению, у обработчика inv не предусмотрены дополнительные параметры, но если вы вспомните, что команда STEAD3 раскладывается на компоненты при ее обработке, то поймете следующий код:

```
obj {
    nam = '@compass';
    disp = false;
    inv = function(s)
        local dir = std.cmd[3]
        p("Вы хотите идти по направлению: ", dir)
    end;
}:with {
```

Теперь в переменной dir мы получим; n, s, w или e! Действительно, в std.cmd[1] будет записан 'use', в std.cmd[2] – объект, а в std.cmd[3] – параметр.

Если бы мы вставляли подстановку в тело вывода сцены, команда была бы такой:

```
act @compass,n
```

И был бы вызван обработчик act. В обработчике act предусмотрены дополнительные аргументы, поэтому в таком случае мы могли бы просто взять параметр из списка параметров к обработчику act. Впрочем, способ с массивом std.cmd сработал бы тоже.

Теперь нам осталось написать код, который делает переход по локации, заданной атрибутом to_направление:

```
obj {
    nam = '@compass';
    disp = false;
    inv = function(s)
        local dir = std.cmd[3]
        local r = std.call(std.here(), 'to_'..dir)
        if not r then
            std.p ([[Нет прохода.]])
        else
```

```
            std.walk(r)
        end
    end;
}:with {
```

Теперь нам осталось только сделать так, чтобы компас попадал в инвентарь и предоставить объект compass для настроек (которые могут в нем появиться при дальнейшем развитии модуля):

```
local std = stead
require ”fmt”

obj {
    nam = ’@compass’;
    disp = false;
    inv = function(s)
        local dir = std.cmd[3]
        local r = std.call(std.here(), dir..’_to’)
        if not r then
            std.p ([[Нет прохода]])
        else
            std.walk(r)
        end
    end;
}:with {
    obj {
        nam = ’$compass’;
        act = function(s, w)
            std.p(fmt.c (”{@compass n| C}\n{@compass w|3} {@compass e|B}\n{@compas
        end;
    }
}

std.mod_start(function(load)
    take ’@compass’
end)
compass = _’@compass’
```

Дальнейшие улучшения компаса могут включать в себя:

- не показывать ссылками те направления, которые недоступны. Для этого надо изменить код $compass.act;
- оформить вывод компаса в виде изображения. Для этого можно с помощью pixels сформировать изображение компаса, разрезать изображение на 9 квадратных изображений, преобразовать из в спрайты и выводить ссылки на них из act.

Данные улучшения предлагается выполнить самостоятельно.

7.2 Модуль клавиатурного ввода

В качестве следующего примера рассмотрим код модуля keyboard. Этот модуль можно скачать с репозитория модулей[1]. Модуль позволяет организовать ввод пользователя с клавиатуры.

Идея модуля состоит в том, что клавиатурный ввод оформлен в виде специальной комнаты, в которой можно выполнить набор текста и вернуться к первоначальной комнате.

Комната, содержащая в себе логику клавиатурного ввода, носит имя @keyboard. Это системный объект, который не уничтожается при рестарте мира.

Для использования клавиатуры используется ссылка на системный объект (см. main3.lua):

```
p [[Как вас {@keyboard "Имя"|зовут}?]];
```

Как видим, в качестве параметра передается информационная строка, которая будет отображена при вводе.

Объект @keboard, соответственно, должен реализовать act обработчик, который перенесет игрока в комнату клавиатурного ввода:

```
act = function(s, w, ...)
    s.title = w or "?"
    s.args = { ... }
    walkin(s)
end;
```

[1] https://github.com/instead-hub/stead3-modules/tree/master/keyboard

Кроме того, что здесь мы меняем название комнаты (title) и делаем переход (walkin), мы также запомнили все дополнительные аргументы, если они передавались в ссылке {@keyboard}. Зачем это нужно, будет ясно позднее.

Итак, осталось реализовать клавиатурный ввод. Для этого мы воспользуемся ссылками кнопок (для того чтобы вводить текст можно было на устройствах без клавиатуры). И одновременно с этим, будем отслеживать нажатия на клавиши с помощью модуля keys (для удобства игроков на компьютерах с клавиатурой).

Формирование ссылок-клавиш осуществляется в decor. При этом, ссылки формируются в виде:

```
row = row..”{@kbdinput ”..vv..”|”..input_esc(a)..”}”..fmt.nb ”  ”;
```

input_esc() – функция, реализованная в модуле, которая экранирует некоторые символы, типа { и }.

Как видим, это снова ссылки на системный объект, но теперь это уже @kbdinput. Этот объект специально создан для обработки событий от клавиш. Прежде чем мы перейдем к нему, рассмотрим вопрос использования модуля keys.

Для использования keys мы должны:

- определить onkey в комнате @keyboard;
- задать keys:filter()

Мы могли бы просто определить в модуле:

```
function keys:filter()
-- логика функции
end
```

Но мы хотим, чтобы наш модуль мог работать совместно с каким то другим применением модуля keys, поэтому при старте игры мы перехватываем старый обработчик keys:filter, заменяя его на свой. А потом, при деинициализации мира возвращаем старый обработчик обратно (когда он уже не нужен).

```
local hooked
local orig_filter
```

```
std.mod_start(function(load)
    if not hooked then
        hooked = true
        orig_filter = std.rawget(keys, 'filter')
        std.rawset(keys, 'filter', std.hook(keys.filter, function(f, s, press, key)
            if std.here().keyboard_type then
                return hook_keys[key]
            end
            return f(s, press, key)
        end))
    end
end)

std.mod_done(function(load)
    hooked = false
    std.rawset(keys, 'filter', orig_filter)
end)
```

Здесь мы видим использование нескольких функций:

- std.rawget(таблица, имя) - получить элемент таблицы без генерации сообщения о несуществующей переменой (работа с переменными на низком уровне);
- std.rawset(таблица, имя, значение) - установить значение элемента таблицы без генерации сообщения о необъявленной переменной;
- std.hook(старая функция, функция перехвата) - создать функцию перехвата.

std.rawget/std.rawset – это способ работы с переменными на самом низком уровне. Например, если бы в mod_start мы просто присвоили keys.filter = std.hook …, то мы получили бы сообщение о том, что мы меняем объект, который не является переменной.

Итак, если клавиши нажимаются в момент нахождения в комнате @keyboard, то мы используем наш фильтр клавиш, если нет – используется перехваченный фильтр клавиш (keys:filter).

Массив hook_keys содержит все те клавиши, которые мы перехватываем.

Объект @kbdinput реализует act, который занимается обработкой вводимых символов. Когда текст набран и ввод подтвержден клавишой ввод или соответствующей ссылкой выполняется следующий код:

```
walkback();
return std.call(std.here(), 'onkbd', _'@keyboard'.text, std.unpack(_'@keyboard'.ar
```

Как видим, происходит вызов обработчика onkbd в комнате из которой мы пришли. В качестве первого параметра передается введенный текст. Затем передаются все те параметры, которые пользователь задал в ссылке @keyboard. Эти параметры помогут идентифицировать поле ввода, если мы хотим использовать несколько полей ввода в одной комнате.

Здесь используются следующие функции:

- std.unpack(таблица) – превратить таблицу в набор аргументов;
- std.call(объект, метод, аргументы) – вызвать обработчик объекта instead.

Теперь вы можете самостоятельно разобраться в остальных деталях модуля @keyboard.