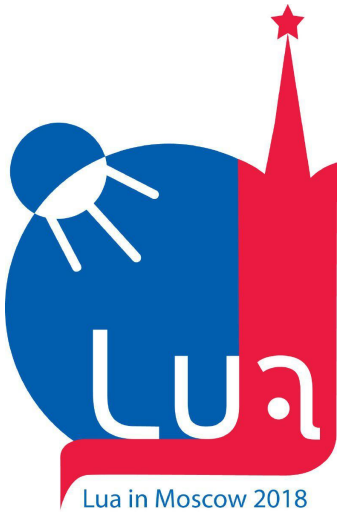
IPONWEB

Зачем и как мы добавляли новые функции в нашу реализацию Lua

Антон Солдатов, IPONWEB

24 марта 2018 г.



Lua в IPONWEB

- Технологические решения в сфере онлайн-рекламы (real-time bidding, RTB)
- Используем Lua более 10 лет для разработки бизнес-логики
- С 2015 г. – собственная реализация Lua

О чём пойдёт речь?

- Неизменяемые объекты
- Прерывание корутин по тайм-ауту

Неизменяемые объекты

Неизменяемые объекты

```
local v = ...
v = ujit.immutable(v)
```

Неизменяемые объекты

```
ujit.immutable(_G)
```

Mechanisms, not Policy

```
function readOnly(t)
    local proxy = {}
    local mt = {        -- create metatable
        __index = t,
        __newindex = function (t,k,v)
            error("attempt to update a read-only table", 2)
        end
    }
    setmetatable(proxy, mt)
    return proxy
end
```

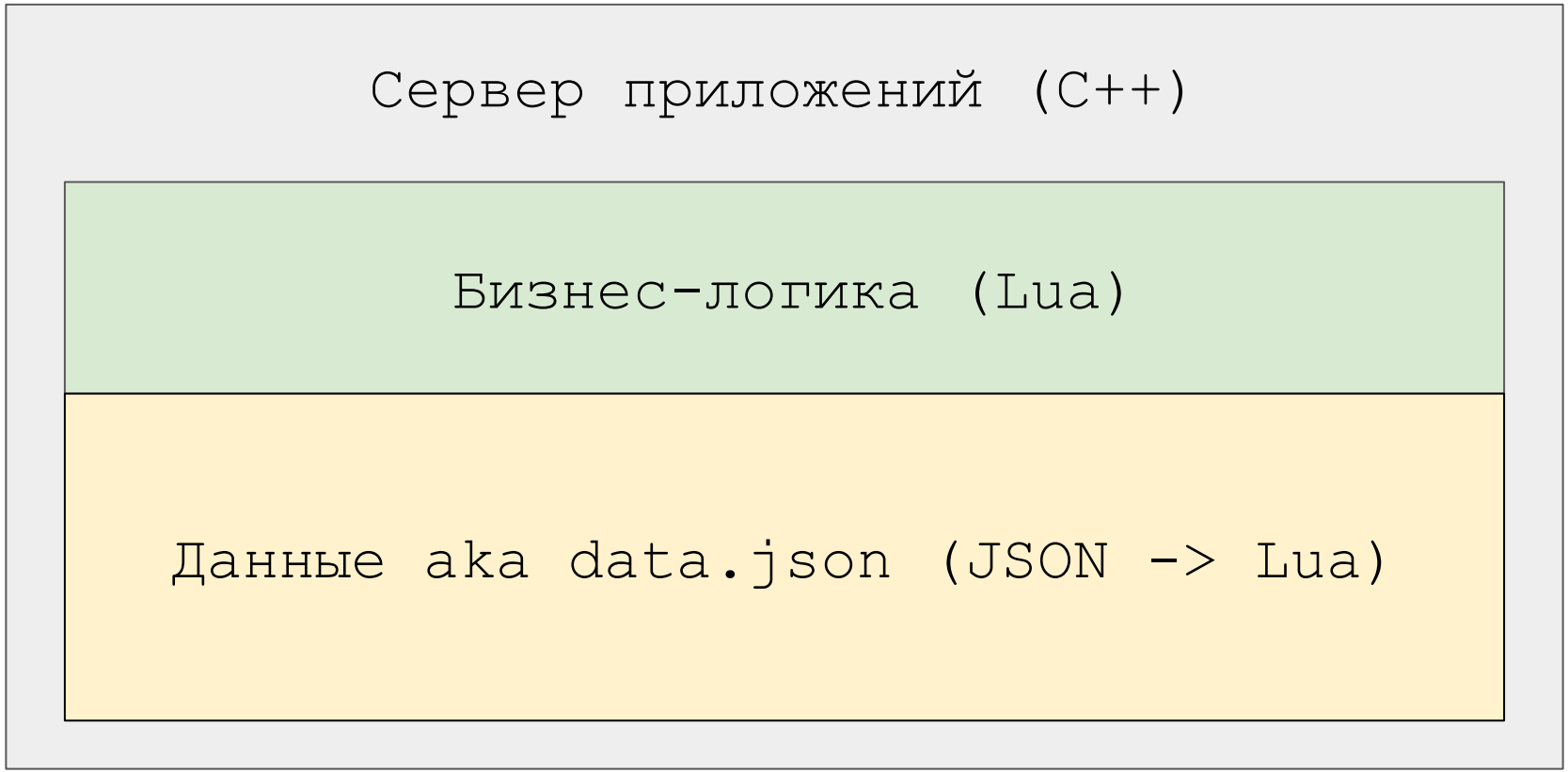
Mechanisms, not Policy

```
local days = readOnly{
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
}
print(days[1])      --> Sunday
days[2] = "Noday"
stdin:1: attempt to update a read-only table
```

Немного контекста

- Неизменяемость объектов "выросла" из другой функциональности, т.н. sealed-объектов
- sealed-объекты решали проблему неоптимальной работы сборщика мусора

Архитектура сервера приложений

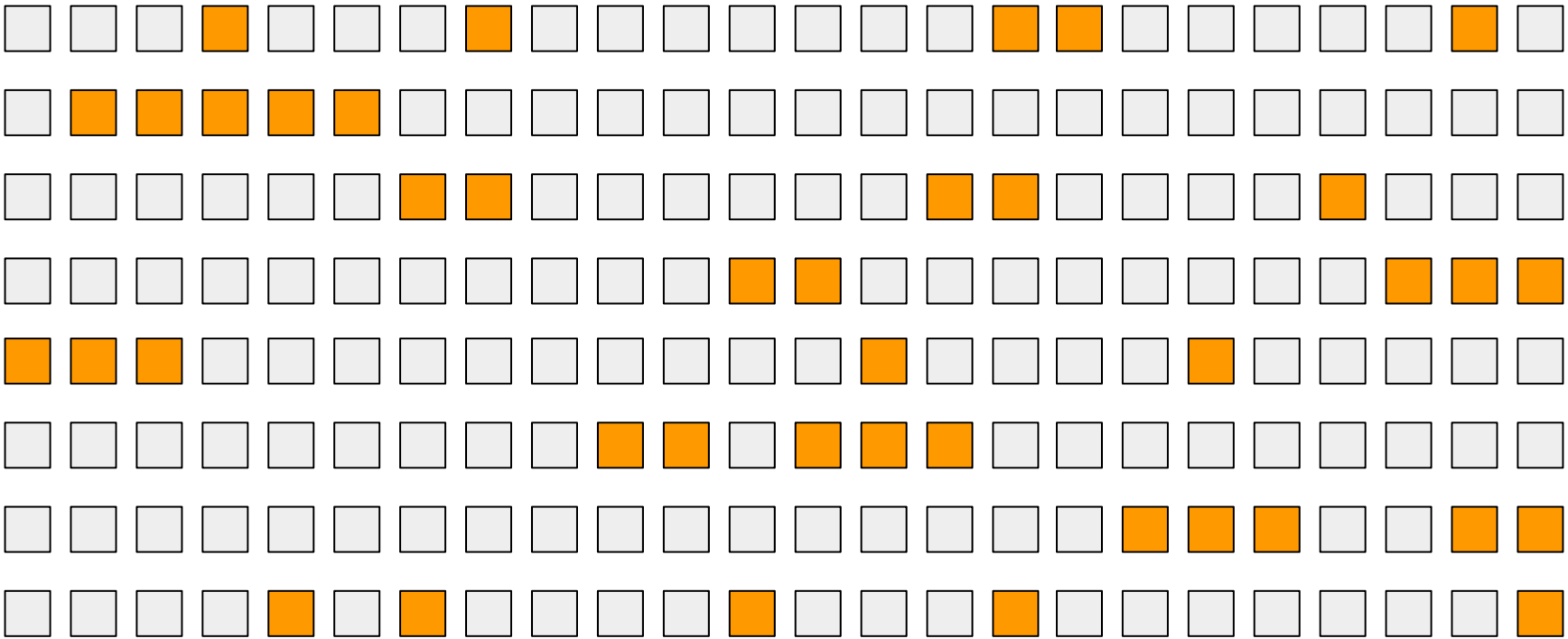


Свойства данных из `data.json`

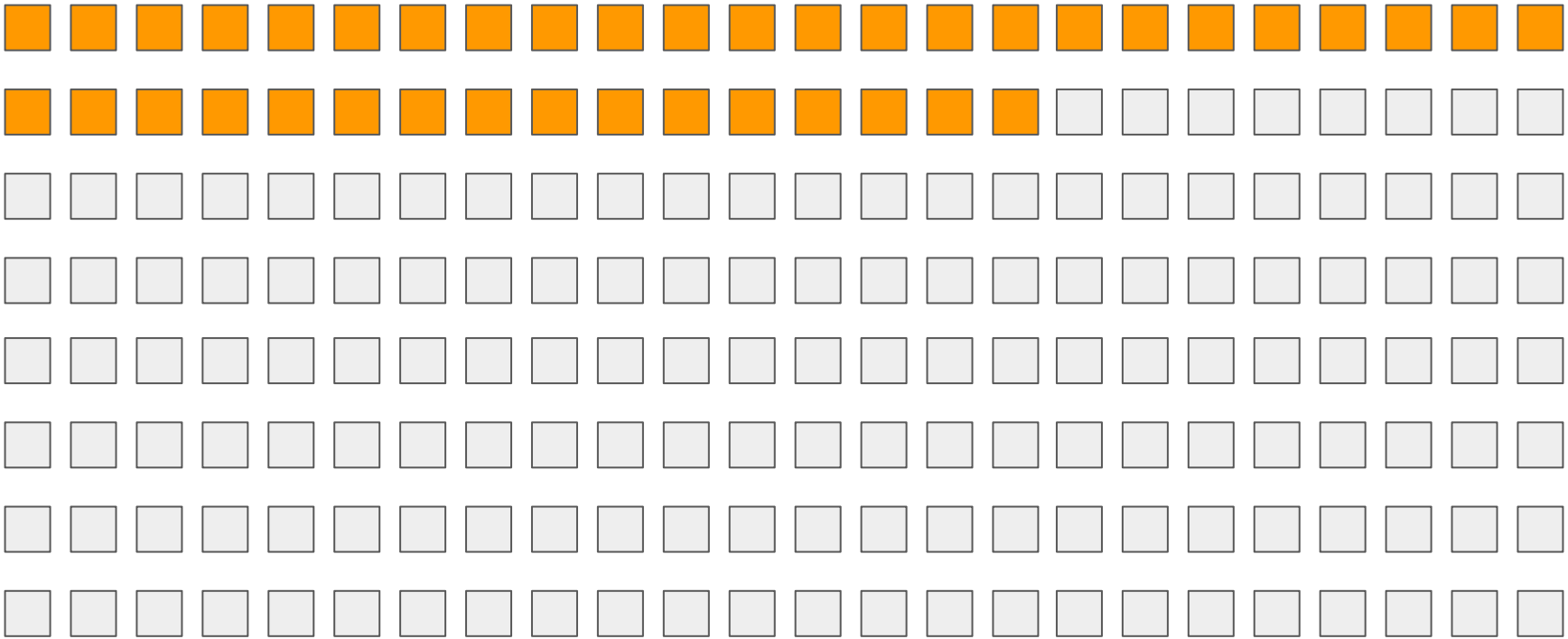
- Загружаются на старте
- Постоянно доступны бизнес-логике
- Не изменяются со временем

Потребление памяти Lua-интерпретатором

Распределение объектов в памяти



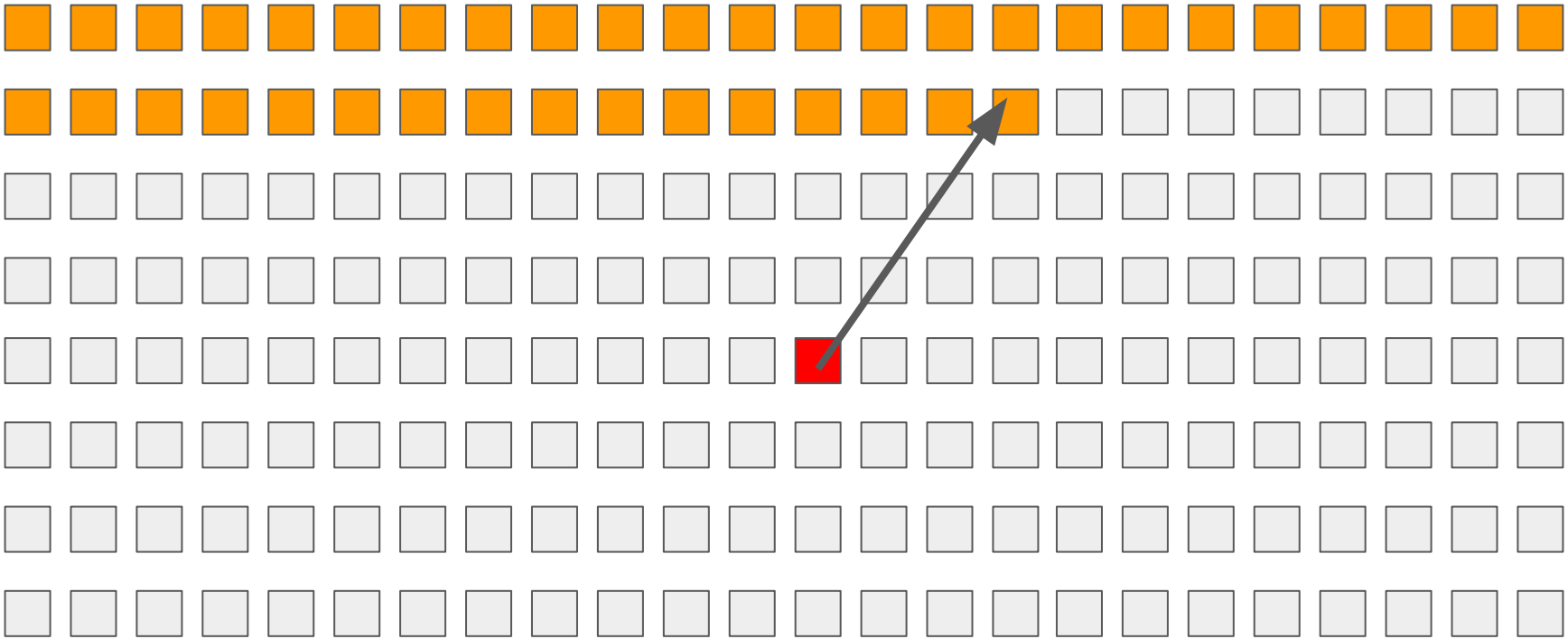
`ujit.seal(data)`



Свойство sealed-объектов

- Сборщик мусора просматривает объекты до первого "запечатанного"
- Следовательно, все sealed-объекты должны быть **неизменяемы**

`ujit.seal(data)`



```
seal = "seal per se" + immutable
```

`immutable`: Типы данных

`nil`, `boolean`, `number`, `string`, `function`, `userdata`	no-op
`thread`	Ошибка времени исполнения
`table`	Рекурсивная неизменяемость

`immutable`: Простые структуры

```
local t = ujit.immutable({foo = "bar"})

-- All of the following throw:

t.new = "baz" -- Add
t.foo = "baz" -- Modify
t.foo = nil   -- Remove
```

`immutable`: Сложные структуры (пример 1)

```
local t = ujit.immutable({{foo = "bar"}})

-- All of the following throw:

t[1].new = "baz" -- Add
t[1].foo = "baz" -- Modify
t[1].foo = nil   -- Remove
```

`immutable`: Сложные структуры (пример 2)

```
local t = {
    {foo = "bar"},
    ujit.immutable({foo = "bar"}),
}


t[1].foo = nil -- OK
t[2].foo = nil -- ERROR
```

`immutable`: Стандартная библиотека

```
local t = ujit.immutable({1, 8, 1942})

-- All of the following throw:

rawset(t, 1, 2)
table.insert(t, 2018)
table.remove(t, 3)
table.sort(t)
```

`immutable`: C API

```
luaE_immutable(L, idx);

/* All of the following throw: */

lua_setfield(L, idx, "foo");
lua_settable(L, idx);
lua_rawset(L, idx);
```

`immutable`: Таблицы и метатаблицы

- Неизменяемые таблицы могут иметь метатаблицы
  - `__newindex` не вызывается
- Неизменяемые таблицы не могут менять свои метатаблицы
- Метатаблицы неизменяемых таблиц неизменяемы
  - Возможны сайд-эффекты

`immutable`: Метатаблицы

```
local t = setmetatable({foo = "bar"}, {...})
ujit.immutable(t)

-- All of the following throw:

setmetatable(t, {})  -- or debug.setmetatable
setmetatable(t, nil) -- or debug.setmetatable
getmetatable(t).foo = "bar"
```

`immutable`: Прочие свойства

- Неизменяемость нельзя "отключить"

`immutable`: Прочие свойства

- Неизменяемость нельзя "отключить"
- Не влияет на сборку мусора

`immutable`: Прочие свойства

- Неизменяемость нельзя "отключить"
- Не влияет на сборку мусора
- Поддержка JIT-компилятором

`immutable/seal`: Результат разделения

- Две новых функции

`immutable/seal`: Результат разделения

- Две новых функции
- Coupling в кодовой базе уменьшился

`immutable/seal`: Результат разделения

- Две новых функции
- Coupling в кодовой базе уменьшился
- Производительность улучшилась

`immutable/seal`: Результат разделения

- Две новых функции
- Coupling в кодовой базе уменьшился
- Производительность улучшилась
- `ujit.immutable` – практически бесплатна

Прерывание корутин по тайм-ауту

```
local co = coroutine.create(function ()
                for i=1, 10 do
                  print("co", i)
                  coroutine.yield()
                end
end)
coroutine.resume(co)    --> co  1
coroutine.resume(co)    --> co  2
...
coroutine.resume(co)    --> co  10
```

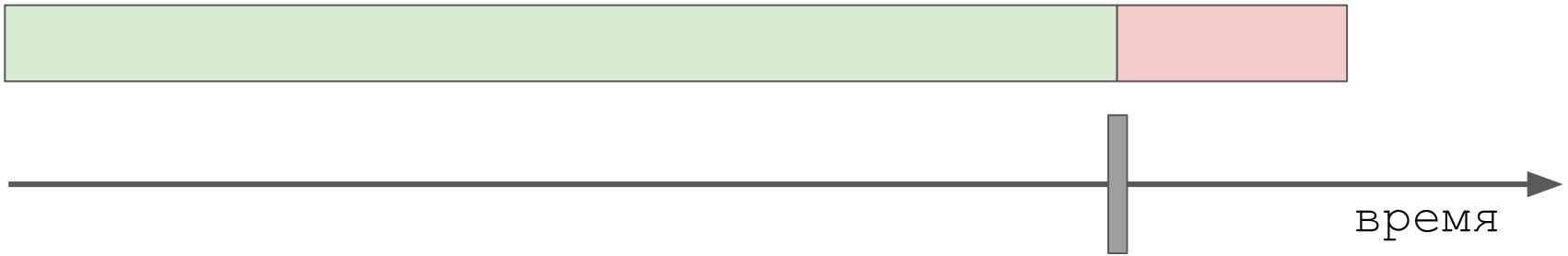
Как мы используем корутины?

- 1 запрос = 1 корутина
- Корутины возвращают управление на блокирующих операциях
- Менеджер корутин выбирает, какую корутину исполнять следующей

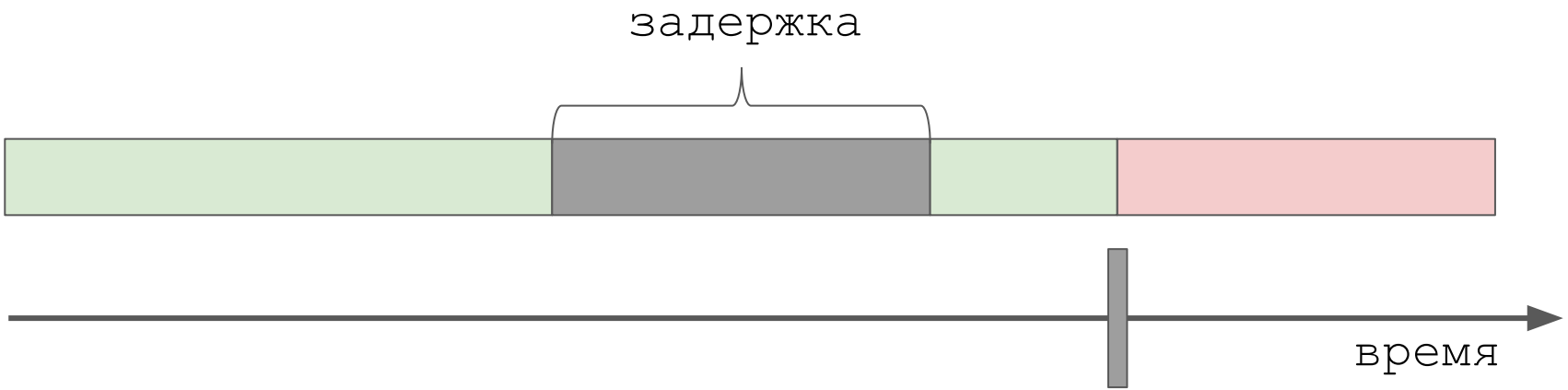
Почему перестало хватать кооперативности?

- Жёсткие временные ограничения на обработку запроса (100-120 мс)
- Прекратим обрабатывать "просроченные" запросы как можно раньше – сэкономим ресурсы

Проблема: Бизнес-логика исполняется долго

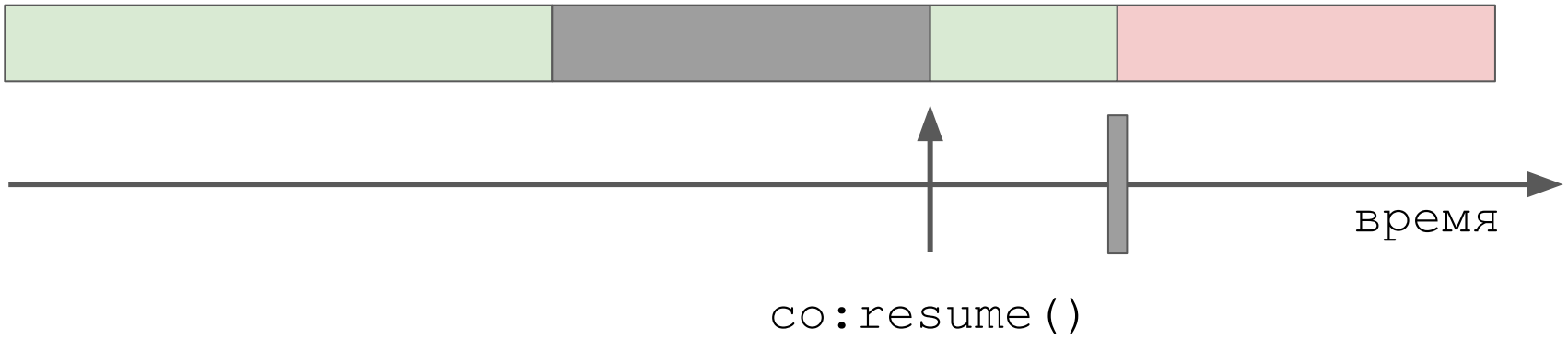


Проблема: Задержки

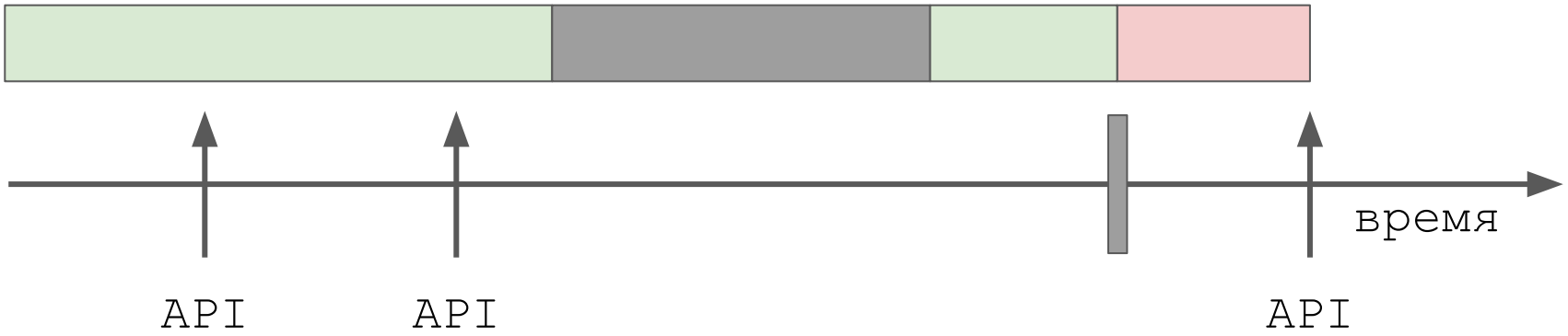


Гранулярность проверок

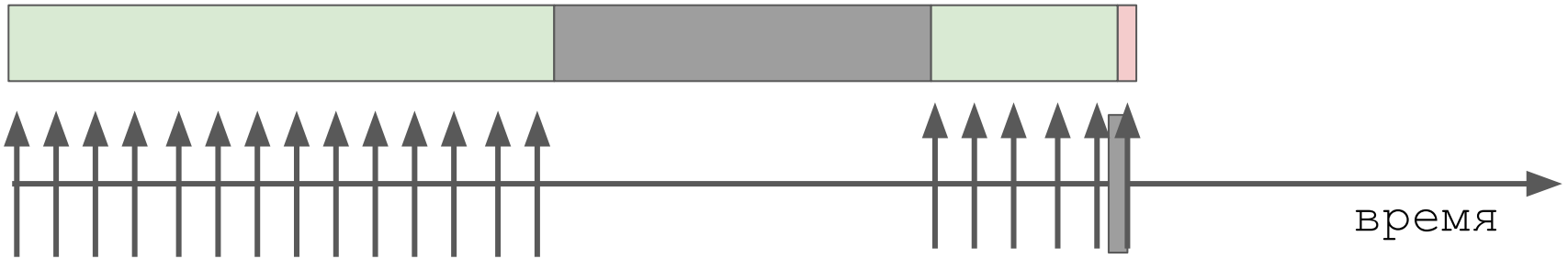
Проверки при возобновлении корутины



Проверки в момент API-вызовов



Проверки внутри виртуальной машины



Прерывания: Как это должно работать?

```
struct timeval timeout = {...};

if (luaE_settimeout(L, &timeout) != 0)
        /* Handle error */;

int status = lua_resume(L, narg);
if (status == LUA_TIMEOUT) {
        /* Handle coroutine expiration */
}
```

Прерывания: Пользовательский обработчик

```
/* lua_CFunction */
static int cb_timeout(lua_State *L)
{
        debug_traceback(L);
        write_logs(...);
        send_default_response(...);
        return 0;
}

luaE_settimeoutf(L, cb_timeout);
```

Прерывания: Особые случаи

- Обработчики ошибок (2-й аргумент `xpcall`)

Прерывания: Особые случаи

- Обработчики ошибок (2-й аргумент `xpcall`)
- Хуки (`debug.{get,set}hook`)

Прерывания: Особые случаи

- Обработчики ошибок (2-й аргумент `xpcall`)
- Хуки (`debug.{get,set}hook`)
- Метаметод `__gc`

Прерывания: Особые случаи

- Обработчики ошибок (2-й аргумент `xpcall`)
- Хуки (`debug.{get,set}hook`)
- Метаметод `__gc`
- JIT-компилятор

Прерывания: Обработка тайм-аута

- Проверяем, разрешена ли проверка тайм-аутов
- В случае тайм-аута:
  - Исполняем пользовательский обработчик
  - Возвращаем управление в точку вызова

    `lua_resume`

Прерывания: Результаты

- Возможность не тратить время на ненужные вычисления

Прерывания: Результаты

- Возможность не тратить время на ненужные вычисления
- Хорошая гранулярность проверок

Прерывания: Результаты

- Возможность не тратить время на ненужные вычисления
- Хорошая гранулярность проверок
- Практически нулевые накладные расходы

А ещё...

- Профилировщик
- Анализатор дампов JIT-компилятора (bit.ly/dumpanalyze)
- Разделение read-only данных между разными экземплярами Lua VM
- Быстрый C API для итерации по таблицам

Как добавлять новые функции?

- Не меняйте синтаксис языка

Как добавлять новые функции?

- Не меняйте синтаксис языка
- Не меняйте стандартную библиотеку

Как добавлять новые функции?

- Не меняйте синтаксис языка
- Не меняйте стандартную библиотеку
- Следите за префиксами функций в C API

Как добавлять новые функции?

- Не меняйте синтаксис языка
- Не меняйте стандартную библиотеку
- Следите за префиксами функций в C API
- Документируйте изменения (с учётом версий)

Выводы

- Внесение изменений в реализацию языка приносит свои выгоды, но весьма трудоёмок
- Взвешивайте все "за" и "против"
- Помните о сохранении совместимости

Спасибо! Вопросы?

Ссылки

- bit.ly/iow-hl-2016
- bit.ly/iow-lim-2017-03
- bit.ly/iow-hl-2017

- bit.ly/dumpanalyze
- asoldatov@iponweb.net, t.me/igelhaus