Sphinx with Markdown Documentation

Release 0.1.0

Marijn van der Zee

Mar 11, 2017

Contents

1 Question: Using sphinx with Markdown instead of RST **3**

2 Marijn’s Answer **5**

3 Beni’s very comprehensive answer **7**

4 Concurrency Model **9**

4.1 What is Concurrency? 9

4.2 Concurrent Activation in NPL 9

4.3 NPL Message Scheduling 12

4.4 Language Compare: The Mental Picture 13

5 ParaEngine **15**

6 Test **17**

7 Indices and tables **19**

This is a Sphinx documentation project that shows how you could use Markdown in Sphinx.

Contents:

CHAPTER 1

Question: Using sphinx with Markdown instead of RST

I hate RST but love sphinx. Is there a way that sphinx reads markdown instead of reStructuredText?

CHAPTER 2

Marijn's Answer

This is my answer, taken from StackOverflow

You can use Markdown and reStructuredText in the same Sphinx project. How to do this is succinctly documented on Read The Docs. Install recommonmark (`pip install recommonmark`) and then edit `conf.py`:

```
from recommonmark.parser import CommonMarkParser

source_parsers = {
    '.md': CommonMarkParser,
}

source_suffix = ['.rst', '.md']
```

CHAPTER 3

Beni's very comprehensive answer

This is Beni's answer, taken from StackOverflow

The "proper" way to do that would be to write a docutils parser for markdown. (Plus a Sphinx option to choose the parser.) The beauty of this would be instant support for all docutils output formats (but you might not care about that, as similar markdown tools already exist for most). Ways to approach that without developing a parser from scratch:

- You could cheat and write a "parser" that uses Pandoc to convert markdown to RST and pass that to the RST parser :-).
- You can use an existing markdown->XML parser and transform the result (using XSLT?) to the docutils schema.
- You could take some existing python markdown parser that lets you define a custom renderer and make it build docutils node tree.
- You could fork the existing RST reader, ripping out everything irrelevant to markdown and changing the different syntaxes (this comparison might help)...EDIT: I don't recommend this route unless you're prepared to heavily test it. Markdown already has too many subtly different dialects and this will likely result in yet-another-one...

UPDATE: https://github.com/sgenoud/remarkdown is a markdown reader for docutils. It didn't take any of the above shortcuts but uses a Parsley PEG grammar inspired by peg-markdown. Doesn't yet support directives.

UPDATE: https://github.com/rtfd/recommonmark and is another docutils reader, natively supported on ReadTheDocs. Derived from remarkdown but uses the CommonMark-py parser. Doesn't support directives, but can convert more or less natural Markdown syntaxes to appropriate structures e.g. list of links to a toctree. For other needs, an ` ```eval_rst ` fenced block lets you embed any rST directive.

In **all** cases, you'll need to invent extensions of Markdown to represent Sphinx directives and roles. While you may not need all of them, some like `.. toctree::` are essential.This I think is the hardest part. reStructuredText before the Sphinx extensions was already richer than markdown. Even heavily extended markdown, such as pandoc's, is mostly a subset of rST feature set. That's a lot of ground to cover!

Implementation-wise, the easiest thing is adding a generic construct to express any docutils role/directive. The obvious candidates for syntax inspiration are:

- Attribute syntax, which pandoc and some other implementations already allow on many inline and block constructs. For example `` `foo`{.method} `` -> `` `foo`:method: ``.
- HTML/XML. From `<span class="method">foo</span>` to the kludgiest approach of just inserting docutils internal XML!
- Some kind of YAML for directives?

But such a generic mapping will not be the most markdown-ish solution... Currently most active places to discuss markdown extensions are https://groups.google.com/forum/#!topic/pandoc-discuss, https://github.com/scholmd/scholmd/

This also means you can't just reuse a markdown parser without extending it somehow. Pandoc again lives up to its reputation as the swiss army knife of document conversion by supporting custom filtes. (In fact, if I were to approach this I'd try to build a generic bridge between docutils readers/transformers/writers and pandoc readers/filters/writers. It's more than you need but the payoff would be much wider than just sphinx/markdown.)

Alternative crazy idea: instead of extending markdown to handle Sphinx, extend reStructuredText to support (mostly) a superset of markdown! The beauty is you'll be able to use any Sphinx features as-is, yet be able to write most content in markdown.

There is already considerable syntax overlap; most notably link syntax is incompatible. I think if you add support to RST for markdown links, and `###`-style headers, and change default `` `backticks` `` role to literal, and maybe change indented blocks to mean literal (RST supports `> ...` for quotations nowdays), you'll get something usable that supports most markdown.

CHAPTER 4

Concurrency Model

This is an in-depth topic about NPL's concurrency model and design principles. It also compares it with other similar models in popular languages, like erlang, GO, Scala(java), etc.

What is Concurrency?

The mental picture of all computer languages is to execute in sequential order. Concurrency is a language feature about writing code that runs concurrently. Traditional way of doing this is via threads and locks, which is very troublesome to write. The Actor Model, which was first proposed by Carl Hewitt in 1973, takes a different approach to concurrency, which should avoid the problems caused by threading and locking.

There are many implementations of concurrency model in different languages, they differ both in performance under different use cases, and in the programmers' mental picture when writing concurrent code.

> NPL uses a hybrid approach, which give you the ability to run tens of thousands of tasks in a single thread or across multiple threads. More importantly, you do not need to write any code to spawn a virtual process or write error-prone message loops. In short, NPL is very fast, scalable and give programmers a mental picture that is close to neurons in the brain.

Concurrent Activation in NPL

Preemptive vs Non-Preemptive File Activation

By default NPL activate function is non-preemptive, it is the programmer's job to finish execution within reasonable time slice. The default non-preemptive mode gives the programmer full control of how code is executed and different neuron files can easily share data using global tables in the same thread.

On the other hand, NPL also allows you to do preemptive activation, in which the NPL runtime will count virtual machine instructions until it reaches a user-defined value (such as 1000) and then automatically yield(pause) the activate function. The function will be resumed automatically in the next time slice. NPL time slice is by default about 16ms (i.e. 60FPS).

To make file activate function preemptive, simply pass a second parameter `{PreemptiveCount,MsgQueueSize,[filename|name],clear}` to `NPL.this` like below:

- PreemptiveCount: is the number of VM instructions to count before it yields. If nil or 0, it is non-preemptive. Please note JIT-compiled code does not count as instruction by default, see below.
- MsgQueueSize: Max message queue size of this file, if not specified, it is same as the NPL thread's message queue size.

- filename|name: virtual filename, if not specified, the current file being loaded is used.
- clear: clear all memory used by the file, including its message queue. Normally one never needs to clear. A neuron file without messages takes less than 100 bytes of memory (mostly depending on the length's of its filename)

```
-- this is a demo of how to use preemptive activate function.
NPL.this(function()
    local msg = msg; -- important to keep a copy on stack since we go preemptive.
    local i=0;
    while(true) do
        i = i + 1;
        echo(tostring(msg.name)..i);
        if(i==400) then
            error("test runtime error");
        end
    end
end, {PreemptiveCount = 100, MsgQueueSize=10, filename="yourfilename.lua"});
```

You can test your code with:

```
NPL.activate("yourfilename.lua", {name="hi"});
```

Facts about preemptive activate function:

- It allows you to run tens of thousands of jobs concurrently in the same system thread. Each running job has its own stack and the memory overhead is about 450bytes. A neuron file without pending messages takes less than 100 bytes of memory (mostly depending on the length's of its filename). The only limitation to the number of concurrent jobs is your system memory.
- There is a slight performance penalty on program speed due to counting VM instructions.
- With preemptive activate function, the programmer should pay attention when making changes to shared data in the thread, since your function may be paused at any instruction. The golden rule here is never make any changes to shared data, but use messages to exchange data.
- C/C++ API call is counted as one instruction, so if you call ParaEngine.Sleep(10), it will block all concurrent jobs on that NPL thread for 10 seconds.
- Code in async callbacks (such as timer, remote api call) in activate function are NOT preemptive. Because callbacks are invoked from the context of other concurrent activate functions.

Test Cases and Examples:

see also `script/ide/System/test/test_concurrency.lua` for more tests cases.

```
local test_concurrency = commonlib.gettable("System.Core.Test.test_concurrency");

function test_concurrency:testRuntimeError()
    NPL.this(function()
        local msg = msg; -- important to keep a copy on stack since we go preemptive.
        local i=0;
        while(true) do
            i = i + 1;
            echo(tostring(msg.name)..i);
            if(i==40) then
                error("test runtime error");
            end
        end
    end, {PreemptiveCount = 100, MsgQueueSize=10, filename="tests/testRuntimeError"});
    NPL.activate("tests/testRuntimeError", {name="1"});
    NPL.activate("tests/testRuntimeError", {name="1000"});
end

function test_concurrency:testLongTask()
    NPL.this(function()
        local msg = msg; -- important to keep a copy on stack since we go preemptive.
        local i=0;
        while(true) do
            i = i + 1;
            echo(i);
        end
    end, {PreemptiveCount = 100, MsgQueueSize=10, filename="tests/testLongTask"});
    NPL.activate("tests/testLongTask", {name="1"});
end

function test_concurrency:testMessageQueueFull()
    NPL.this(function()
        local msg = msg; -- important to keep a copy on stack since we go preemptive.
        local i=0;
        for i=1, 1000 do
            i = i + 1;
        end
        echo({"finished", msg});
    end, {PreemptiveCount = 100, MsgQueueSize=3, filename="tests/testMessageQueueFull
↪"});
    for i=1, 10 do
        NPL.activate("tests/testMessageQueueFull", {index=i});
    end
    -- result: only the first three calls will finish.
end

function test_concurrency:testMemorySize()
    __rts__:SetMsgQueueSize(100000);
    for i=1, 10000 do
        NPL.this(function()
            local msg = msg; -- important to keep a copy on stack since we go␣
↪preemptive.
            for i=1, math.random(1,1000) do
                msg.i = i;
            end
            echo(msg);
        end, {PreemptiveCount = 10, MsgQueueSize=1000, filename="tests/testMemorySize
↪"..i});
        NPL.activate("tests/testMemorySize"..i, {index=i});
    end
    -- TODO: use a timer to check when it will finish.
end

function test_concurrency:testThroughput()
    __rts__:SetMsgQueueSize(100000);
    for i=1, 10000 do
        NPL.this(function()
            local msg = msg;
            while(true) do
                echo(msg)
            end
        end, {PreemptiveCount = 10, MsgQueueSize=3, filename="tests/testThroughput"..
↪i});
        NPL.activate("tests/testThroughput"..i, {index=i});
    end
end
```

NPL Message Scheduling

Each NPL runtime can have one or more NPL states/threads (i.e. real system threads). Each NPL state has a single message queue for input and output with other NPL threads or remote processes. Programmer can set the maximum size of this queue, so when it is full, messages are automatically dropped.

On each time slice, NPL state will process `ALL` messages in its message queue in priority order.

- If a message belongs to a non-preemptive file, it will invoke its activate function immediately.
- If a message belongs to a preemptive file, it will remove the message from the queue and insert it to the target file's message queue, which may has a different message queue size. If message queue of the file is full, it will drop the message immediately.

In another native thread timer, all active preemptive files (with pending/half-processed messages) will be processed/resumed in `preemptive` way. I.e. we will count VM(virtual machine) instructions for each activate function and pause it if necessary.

> Note: all TCP/IP network connections are managed by a single global network IO thread. In this way you can have tens of thousands of live TCP connections without using much system memory. A global NPL dispatcher automatically dispatch incoming network messages to the message queue of the target NPL state/thread. Normally, the number of NPL threads used is close to the number of CPU cores on the computer.

Causions on Preemptive Programming

Since preemptive code and non-preemptive code can coexit in the same NPL state (thread). It is the programmers' job to ensure preemptive code does not modify global data.

- Note: when NPL debugger is attached, all preemptive code are paused to make debugging possible.
- Also, please note that JIT-compiled code does NOT call hooks by default. Either disable the JIT compiler or edit src/Makefile: `XCFLAGS= -DLUAJIT_ENABLE_CHECKHOOK` This comes with a speed penalty even hook is not set. Our recommendation is to call `dummy()` or any NPL API functions and count that.

Priority of Concurrent Activation

PreemptiveCount is the total number of instructions (or Byte code) executed in an activate function before we pause it. In NPL, `PreemptiveCount` can be specified per activate function, thus giving you fine control over how much computation each activate function can run in a given time slice.

The NPL scheduler for preemptive activation file will guarantee each function will run precisely PreemptiveCount instructions after PreemptiveCount * total_active_process_count instructions every time slice.

So there is absolutely no dead-lock conditions in our user-mode `preemptive` scheduler. The only thing that may break your application is running out of memory. However, each active (running) function only take 400-2000 bytes according to usage. even 1 million concurrently running jobs takes only about 1GB memory. If only half of those jobs are busy doing calculation on average, you get only a little over 500MB memory usage.

Language Compare: The Mental Picture

We will compare concurrency model in several languages in terms of programmer's mental picture and implementation.

Erlang

In erlang, one has to manually call `spawn` function to create a user-mode virtual process. So it is both fast and memory efficient to create millions of erlang processes in a single thread. Erlang simulates `preemptive` scheduling among those processes by counting byte code executed. Erlang processes are currently scheduled on a reduction count basis as described here and here. One reduction is roughly equivalent to a function call. A process is allowed to run until it pauses to wait for input (a message from some other process) or until it has executed 1000 reductions. A process waiting for a message will be re-scheduled as soon as there is something new in the message queue, or as soon as the receive timer (receive ... after Time -> ... end) expires. It will then be put last in the appropriate queue. Erlang has 4 scheduler queues (priorities): 'max', 'high', 'normal', and 'low'.

- Pros: In the viewpoint of programmers, erlang process is `preemptive`. Underneath it is not true `preemptive`, but as along as it does not call `C` functions, the scheduler is pretty accurate.
- Cons: The programmers need to manually name, create and manage the process's message loop (such as when message queue grows too big).

Comparison:

- Similarity:
  - Both NPL and erlang copies messages when sending them
  - Both support `preemptive` concurrent scheduler.
  - Both use user mode code to simulate processes, so that there is almost no limit to the total number of asynchronous entities created.
- Differences:
  - Erlang is `preemptive`; NPL can be not `preemptive` and `non-preemptive`.
  - NPL use source code file name as the entity name, and it does not require the programmer to create and manually write the message loop. Thus the mental picture is `each NPL file has a hidden message loop`.
  - Erlang does not give you explicit control over which real system thread your process is running (instead, it automatically does it for you). In NPL, each neuron file can be explicitly instanced on one or more system-level thread.
  - Erlang scheduler does not give you control over Instruction Count per process before it is preemptied. In NPL, one can specify different instruction count per activate function, thus giving you a much finer control over priority.
  - In NPL we count each C API as a single instruction, while Erlang tries to convert C API to the number of ByteCode executed in the same time.

Go

To me, GO's concurrency model is a wrapper of C/C++ threading model. Its syntax and library can greatly simply the code (if written in C/C++). It uses real system threads, so you can not have many concurrent jobs like erlang or NPL.

Comparison:

- NPL can be written in the same fashion of GO, because it give you explicit control over real system-level thread.

Scala

Scala is a meta language over higher general purpose language like java. Scala is like a rewrite of Erlang in java. Its syntax is in object-oriented fashion with a mixture of Erlang's functional style. So please refer to erlang for comparision. However, Scala is not-preemptive as erlang, it relies on the programmer to yield in its event based library, while NPL supports `preemptive` mode in additional to `non-preemptive` mode.

Final Words

- Unlike erlang, go, scala, NPL is a dynamic and weak type language. In NPL, it is faster to invoke C/C++ code, its byte code is as fast as most strongly typed languages. See NPLPerformance for details.

CHAPTER 5

ParaEngine

namespace **`ParaEngine`**

Functions

int **`my_rand`**()

class **`IObjectScriptingInterface`**

Public Functions

bool **`AddScriptCallback`**(int *func_type*, **const** string &*script_func*)

add a new call back handler. it will override the previous one if any.

Parameters

- `script_func`: format is [filename];[scode] where file name can be any NPL address, scode is a short code sequence to execute. it may has the same format as the GUI event handler. e.g. ";function1()" : calling a global function "(gl)script/character/npc.lua;npc.on_click()" : load script if not loaded before and then calling a global function if this is "", *RemoveScriptCallback()* will be called.

IObjectScriptingInterface::ScriptCallback ***`GetScriptCallback`**(int *func_type*)

return NULL if there is no associated script.

bool **`RemoveScriptCallback`**(int *func_type*)

remove a call back handler

struct **`ScriptCallback`**

#include <IObjectScriptingInterface.h> the game object will keep a list of script call backs.

Public Functions

void **`SetScriptFunc`**(**const** std::string &*script*)

set the script function

int **`ActivateAsync`**(**const** std::string &*code*)

activate the script callback with the given code.

int **`ActivateLocalNow`**(**const** std::string &*script*)

activate the script callback locally now. when function returns, the script has returned.

Public Members

int **`func_type`**
function type

std::string **`script_func`**
the NPL file and function to be activated. Its format is similar to GUI events. e.g. "(gl)script/NPC1.lua;NPC1.On_Click();"

unsigned int **`m_nLastTick`**
last time this function is called.

CHAPTER 6

Test

test search engine

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index

I

IObjectScriptingInterface::AddScriptCallback (C++ function), 15
IObjectScriptingInterface::GetScriptCallback (C++ function), 15
IObjectScriptingInterface::RemoveScriptCallback (C++ function), 15

P

ParaEngine (C++ type), 15
ParaEngine::IObjectScriptingInterface (C++ class), 15
ParaEngine::IObjectScriptingInterface::ScriptCallback (C++ class), 15
ParaEngine::IObjectScriptingInterface::ScriptCallback::ActivateAsync (C++ function), 15
ParaEngine::IObjectScriptingInterface::ScriptCallback::ActivateLocalNow (C++ function), 15
ParaEngine::IObjectScriptingInterface::ScriptCallback::func_type (C++ member), 16
ParaEngine::IObjectScriptingInterface::ScriptCallback::m_nLastTick (C++ member), 16
ParaEngine::IObjectScriptingInterface::ScriptCallback::script_func (C++ member), 16
ParaEngine::IObjectScriptingInterface::ScriptCallback::SetScriptFunc (C++ function), 15
ParaEngine::my_rand (C++ function), 15