nplruntime documentation

Release 0.1.0

Li Xizhi

Apr 01, 2017

Table of Content

CHAPTER 1

Welcome to the NPL wiki!

NPL or Neural Parallel Language is an open source, fast, general purpose scripting language. Its syntax is 100%-compatible with lua. NPL runtime provides essential functionality for building `3D/2D/Server` applications that runs on `windows/Mac/linux/android/iOS`. NPL can mix user-mode preemptive and non-preemptive code. It gives you the concurrency of Erlang and speed of `Java/C++` in the same dynamic and weakly-typed language.

NPLQQ 543474853

Install Guide

```
git clone https://github.com/LiXizhi/NPLRuntime.git
git submodule init && git submodule update
./build_linux.sh
```

See Install Guide for details

Getting Started

Example code

```
local function activate()
   if(msg) then
      print(msg.data or "");
   end
   NPL.activate("(gl)helloworld.lua", {data="hello world!"})
end
NPL.this(activate);
```

Why a New Programming Language?

NPL prototype was designed in 2004, which was then called 'parallel oriented language'. NPL is initially designed to write flexible algorithms that works in a multi-threaded, and distributed environment with many computers across the network. More specifically, I want to have a language that is suitable for writing neural network algorithms, 3d simulation and visualization. Lua and C/C++ affinity was chosen from the beginning.

Usage

To run with GUI, use:

```
npl [parameters...]
```

To run in server mode, use:

```
npls [filename] [parameters...]
```

For example:

```
npls hello.lua
```

CHAPTER 2

Table of Content

CHAPTER 3

NPL

- NPL
- NPL Web: 1080p

CHAPTER 4

What is NPL?

NPL is short for neural parallel language.

- NPL is a general-purpose open-source language. Its syntax is 100% compatible with Lua.
- NPL provides essential functionality for building `3D/2D/Web/Server` applications.
- NPL provides rich C/C++ API and large collections of libraries written in NPL.
- NPL is a single language solution for advanced and interactive GUI, complex opengl/DirectX 3D graphics, scalable webserver and distributed software frameworks. It is cross-platform, high performance, extensible and debuggable.
- NPL is a language system initially designed to operate like the brain. Nodes and connections are ubiquitous; threading and networking logics are hidden from developers.
- NPL can mix user-mode `preemptive` and `non-preemptive` code. It gives you the concurrency of Erlang and speed of `Java/C++` in the same dynamic and weakly-typed language.

  See here for a list of projects written in NPL.

Facts You Should Know

- NPL runtime is written in C/C++. It can utilize the luajit compiler, which dynamically compiles script/byte code into native machine code, thus it can approach native C/C++ performance with some caveats.
- NPL runtime native API is a rich collection of C/C++ functions that is callable from NPL script. It includes networking, graphics, io, audio, assets management, and core NPL/ParaEngine infrastructure. NPL Reference contains its full documentation. However, 99% of time, we never use native API directly in NPL script.
- Instead, we use NPL libraries, which are pure NPL/lua script files, distributed open sourced or in a zip/pkg file. Over 1 million lines of fully-documented NPL script code are available for use. Try to search in the repository of NPL source code, before writing your own.
- NPL libraries are all written in `object oriented` fashion, code is organized in folders and table namespaces.
- Because NPL is written in C/C++, it is cross-platform, extensible and easy to integrate with other thirty-party tools. For examples, NPLRuntime is distributed with following built in plugins: bullet(a robust 3d physics engine), mono(C# scripting module with NPL API), mysql/sqlite(database engine), libcurl(a robust http/ssh client).
- There are two ways to use Lua, one is embedding, the other is extending it. NPL takes the former approach. For example, true preemptive programming would NOT be possible with the second approach. The embedding

approach also allows NPL to expose all of its API and language features to other compiler, for example, each NPL runtime states can host Lua/Mono C#/C++ compiler states all together.

`Hello World` in NPL

Run in command line `npl helloworld.lua`. The content of `helloworld.lua` is like this:

```
print("hello world");
```

Now, there comes a more complicated helloworld. It turns an ordinary `helloworld.lua` file into a neuron file, by associating an `activate` function with it. The file is then callable from any NPL thread or remote computer by its NPL address(url).

```
NPL.activate("(gl)helloworld.lua", {data="hello world!"})
local function activate()
   local msg = msg;
   if(msg) then
      print(msg.data or "");
   end
end
NPL.this(activate);
```

If your code has `*.npl` file extension, you can use or extend NPL syntax via meta programming. For example, above code can be written with NPL syntax in `helloworld.npl` as below.

```
NPL.activate("(gl)helloworld.npl", {data="hello world!"})
this(msg){
   if(msg) then
      print(msg.data or "");
   end
}
```

Why Should I Use NPL?

You may write your next big project in NPL, if you meet any following condition, if not all:

- If you like truly dynamic language like lua.
- If you care about C/C++ compatibility and performance.
- Coming from the C/C++ world, yet looking for a replacement language for java/C#, that can handle heavy load of 2d/3d graphical client or server applications, easier than C/C++.
- Looking for a scripting language for both client and server development that is small and easy to deploy on windows and other platforms.
- Looking for a dynamic language that can be just-in-time compiled and modified at run time.
- Mix user-mode `preemptive` and `non-preemptive` code for massive concurrent computation.

  Note: calling API between C++/scripting runtime, usually requires a context switch which is computationally expensive most language also involves type translation (such as string parameter), managed/unmanaged environment transition (garbage collection), etc. NPL/LUA has the smallest cost among all languages, and with luajit, type translation is even unnecessary. Thus making it ideal to choose NPL when some part of your responsive app needs to be developed in C/C++.

Background

NPL prototype was designed in 2004, which was then called 'parallel oriented language'. In 2005, it was implemented together with ParaEngine, a distributed 3d computer game engine.

Why a New Programming Language?

NPL is initially designed to write flexible algorithms that works in a multi-threaded, and distributed environment with many computers across the network. More specifically, I want to have a language that is suitable for writing neural network algorithms, 3d simulation and visualization. Lua and C/C++ affinity was chosen from the beginning.

Communicate Like The Brain

Although we are still unclear about our brain's exact computational model, however, following fact seems ubiquitous in our brain.

- The brain consists of neurons and connections.
- Data flows from one neuron to another: it is asynchronously, single-direction and without callback. Communication in NPL is the same as above. Each file can become a neuron file to receive messages from other neuron files. They communicate asynchronously without callback. As a result, no lock is required because there is no shared data; making it both simple and fast to write and test distributed algorithms and deploy software in heterogeneous environment.

Compare NPL With Other Languages

- `C/C++`:
  - Pros: They are actually the only popular cross-platform language. NPL Runtime is also written in C++.
  - Cons: But you really need a more flexible scripting language which are dynamically compiled and easier to use.
- `Mono C#`, `Java`:
  - Pros: Suitable for server side or heavy client. Rich libraries.
  - Cons: Deployment on windows platform is big. Writing C++ plugins is hard, calling into C++ function is usually slow. Language is strong typed, not really dynamic.
- `Node.js`, `Electron`:
  - Pros: Suitable for server side or heavy client. HTML5/javascript is popular.
  - Cons: Deployment on the client-side is very big. writing C++ plugins is very hard.
- `Python`, `Ruby`, `ActionScript` and many others:
  - Pros: Popular and modular
  - Cons: Performance is not comparable to C/C++; standard-alone client-side deployment is very big; syntax too strict for a scripting language. Some of them looks alien for C/C++ developers.
- `Lua`:
  - `Pros`: NPL is 100% compatible with it. Really dynamic language, highly extensible by C/C++, and syntax is clear for C/C++ developers.

  - `Cons`: Born as an embedded language, it does not have a standalone general-purpose runtime with native API support and rich libraries for complex client/server side programming. This is where NPL fits in.

- Concurrency Model: in-depth compare with erlang, GO, scala

References

Click the links, if you like to read the old history:

CHAPTER 5

Install NPL Runtime

You can redistribute NPLRuntime side-by-side with your scripts and resource files on windows/linux/iOS, etc. However, this section is only about installing NPL runtime to your development computer and setting up a recommended coding environment for NPL.

Install on Windows

Install from Windows Installer

It is very easy to install on windows. The win32 executable of NPL runtime is currently released in two forms, choose one that is most suitable to you:

- 32bits/64bits most recent development version (recommended for standalone application or module dev). Click to download
- Official stable version released in ParacraftSDK (recommended for paracraft mod dev). See below.

Paracraft is an IDE written completely in NPL. Follow the steps:

- Download ParacraftSDK
  - or you can clone it with following commands (you need git installed):

```
# clone repository & sub modules
git clone https://github.com/LiXizhi/ParaCraftSDK.git
cd ParaCraftSDK
git submodule init
git submodule update
# Go into the repository
cd ParaCraftSDK
# install paracraft
redist\paracraft.exe
# install NPLRuntime
NPLRuntime\install.bat
```

- `redist\paracraft.exe` will install the most recent version of `Paracraft` to `./redist` folder
  - if you failed to install, please manually download from Paracraft and copy all files to `redist` folder.
- Inside `./NPLRuntime/install.bat`
  - The NPL Runtime will be copied from `./redist` folder to `./NPLRuntime/win/bin`
  - The above path will be added to your `%path%` variable, so that you can run npl script from any folder.

Update NPL Runtime to latest version

- If you use NPLRuntime installer, simply download and reinstall it here
- If you use paracraftSDK, we regularly update `paracraft` (weekly) on windows platform, you can update to latest version of NPL Runtime following following steps.
  - run `redist\paracraft.exe` to update paracraft's NPL runtime in `redist` folder
  - run `NPLRuntime\install.bat` to copy NPL runtime executable from `redist` to `NPLRuntime/win/bin` folder
- Compiling from source code is always the latest version (recommended for serious developers), see next section.

On 64bits version

To build 64bits version of NPLRuntime, you need to build from source code (see next section), or download prebuild version from http://www.paracraft.cn.

Our current build target for public users is still 32bits on windows platform.

> Please note that all compiled and non-compiled NPL/lua code can be loaded by both 32bits/64bits version of NPL runtime across all supported platforms (windows/linux/mac/android/ios) etc. However, if you use any C++ plugins (such as mysql, mono, etc), they must be built for each platform and CPU architecture.

Install on Windows From Source

see .travis.yml or our build output for reference

- You need to download and build boost from source code. Version `1.55.0` and `1.60.0` are tested,but the latest version should be fine. Suppose you have unzipped the boost to `D:\boost`, you can build with following command.

```
bootstrap
b2 --build-type=complete
```

and then add a system environment variable called `BOOST_ROOT`, with value `D:\boost`(or where your boost is located), so that our build-tool will find boost for you.

- Download NPLRuntime source code, or clone it by running:

```
git clone https://github.com/LiXizhi/NPLRuntime.git
```

- Download cmake, and build either `server` or `client` version. They share the same code base but with different `cmakelist.txt`; `client` version has everything `server` has plus graphics API.
  - `Server` version: run `NPLRuntime/NPLRuntime/CMakeList.txt` in `NPLRuntime/bin/win32` folder (any folder is fine, but do not use the `cmakelist.txt` source folder), please refer to `./build_win32.bat` for details
  - `Client` version: run `NPLRuntime/Client/CMakeList.txt` in `NPLRuntime/bin/Client` folder (any folder is fine, but do not use the `cmakelist.txt` source folder)
    - **REQUIRED:** Install latest DirectX9 SDK in default location here.
    - There is also a cmake option called `ParaEngineClient_DLL` which will build NPLRuntime as shared library, instead of executable.
- If successful, you will have visual studio solution files generated. Open it with visual studio and build.

- The final executable is in `./ParaWorld/bin/` directory.

How to Debug NPLRuntime C++ source code

When you build NPLRuntime from source code, all binary file(dlls) are automatically copied to `./ParaWorld/bin/` directory. All debug version files have `_d` appended to the end of the dll or exe files, such as `sqlite_d.dll`, etc.

To debug your application, you need to start your application using the executable in `./ParaWorld/bin/`, such as `paraengineclient_d.exe`. You also need to specify a working directory where your application resides, such as `./redist` folder in `ParacraftSDK`. Alternatively, one can also copy all dependent `*_d.dll` files from `./ParaWorld/bin/` to your application's working directory.

Applications written in NPL are usually in pure NPL scripts. So you can download and install any NPL powered application (such as this one), and set your executable's working directory to it before debugging from source code. `ParacraftSDK` also contains a good collection of example NPL programs. Please see the video at [[TutorialWebSite]] for install guide.

Install on Linux

Click here to download precompiled NPL Runtime linux 64bits release.

Under linux, it is highly recommended to build NPL from source code with latest packages and build tools on your dev/deployment machine.

- Download NPLRuntime source code, or clone it by running:

```
git clone https://github.com/LiXizhi/NPLRuntime.git
git submodule init && git submodule update
```

- Install third-party packages and latest build tools.
  - see .travis.yml or our build output for reference (on debian/ubuntu). In most cases, you need `gcc 4.9 or above,cmake,bzip2,boost,libcurl`. Optionally, you can install `mysql,mono,postgresql` if want to build NPL plugins for them.
- Run ./build_linux.sh from the root directory. Be patient to wait for build to complete.

```
./build_linux.sh
```

Please note, you need to have at least 2GB memory to build with gcc, or you need to step up swap memory

- The final executable is in `./ParaWorld/bin64/` directory. A symbolic link to the executable is created in `/usr/local/bin/npl`, so that you can run NPL script from anywhere.

Install on MAC OSX

We use cmake to build under MAC OSX, it is very similar to `install on linux`, but you should use

```
./build_mac_server.sh
```

Currently, only server version is working on mac, client version is still under development.

Editors and Debuggers

You can use your preferred editors that support lua syntax highlighting. NPL Runtime has a built-in web server called [[NPL Code Wiki|NPLCodeWiki]], it allows you to edit and debug via HTTP from any web browser on any platform.

- [[NPL Code Wiki|NPLCodeWiki]]: build-in debugger and editor.

It is though highly recommended that you install NPL Runtime on windows, and use following editors:

- Visual Studio Community Edition: the free version of `visual studio`. This one is what I use.
- Visual Studio Code: a very good free/cross-platform editor based on `Atom`

We have developed several useful open-source NPL language plugins for visual studio products: In visual studio, select `menu::Tools::Extensions`, and search online for `npl` to install following plugins:

- NPL/Lua language service for visual studio: used by thousands of developers worldwide.
  - Download Source code
- NPL Debugger for visual studio: set break points and attach to running NPL process.
  - Download Source code
- NPL/ParaEngine Tools for visual studio: misc tools

Example Code

ParacraftSDK contains many example projects written in NPL.

- See ParacraftSDK wiki site for more details.
- You may now get your hand dirty by this tutorial or continue reading here.

CHAPTER 6

Projects Written in NPL

The following applications are completely written in NPL.

Paracraft

- Started: 2012-present
- Website: www.paracraft.cn
  - ParacraftSDK
  - Self Learning College

Paracraft is a standalone 3d animation software for everyone. It was my exploration of CAD(computer aided design) using the vision brain (i.e. your right brain). Paracraft is an extensible tool that everyone can learn to create videos and computer programs. I am using it to promote NPL language in self learning college to teach students programming.

Wikicraft

- Started: 2016-present
- Website: wikicraft.cn

Project is still in development. It is web site developed using NPL web framework.

Magic Haqi

- Started: 2009-2014 (still operating)
- Website: haqi.61.com
  - user forum
  - user videos

Magic Haqi is a free/paid 3D MMORPG published by taomee in November, 2009, allowing kids to play, create and share 3d worlds. It has over 5 million registered users and tens of thousands of user created works in China. The software is developed and owned by the developers of Paracraft, but has nothing to do with Paracraft. The initial version of paracraft was developed as a module in Magic Haqi. We have valid contract with its publisher taomee for their use of paracraft and its source code.

Magic Haqi2

- Started: 2012-2013 (still operating)
- Website: haqi2.61.com Same as Magic haqi.

Kids Movie Creator

- Started: 2006-2007
- Website: download link

Kids Movie Creator is a very old shareware released in 2006, allowing kids to create 3d world and make movies.

PaperStar

- Started: 2015-present
- Website: ps.61.com

This is a sandbox game developed by taomee.com using NPL and ParacraftSDK.

Projects Sponsored

Our investor will sponsor NPL related projects. If you have an idea, click here to get sponsored. The following is a list of projects being sponsored or open for recruitment.

- NPL 3D1
- NPL UI1
- :heart: NPL1
- :heart: CAD1
- NPL
- Paracraft EarthGIS
- :heart: GITNPL1
- Paracraft BMAX
- NPLMeta
- NPL CAD
- Paracraft
- NPL Replication
-
- 3D
- Npl Voxelizer finished
- NPL finished

- ParaEngine/NPL 3DMAC

Other projects

You are welcome to add your own software here

CHAPTER 7

NPL vs Lua

NPL embeds the small Lua/luajit compiler, and adds tons of native C/C++ API and new programming paradigms, making NPL a general purpose language/runtime.

Lua is an embedded language with the world fastest JIT compiler. The entire lua runtime is written in 6000 lines of code, making it very small and memory efficient to be embedded in another host language/runtime.

Host Languages vs Embedded Compilers

A high-level language contains its syntax, native API and libraries. All dynamic language runtimes must embed one or more compiler implementations in it. Compilers contains low-level implementation of compiling text code to bytecode and then to machine code.

For example, some version of JAVA, python, mono C#, objective C are first compiled into LLVM bytecode, and then the LLVM runtime compiles the bytecode to machine code. LLVM byte code is similar to lua or luajit byte code. Java/Python/MonoC# runtime embeds LLVM compiler in the same way as we embed lua/luajit in NPL. The difference is that LLVM is large and based on C/C++ syntax; but luajit is really small, and based on lua syntax. Languages like Java/Python/C# also have tons of host API in addition to LLVM in the same way of NPL hosting lua. This is why lua is designed to be an embedded language over which other host languages can be devised.

Facts about NPL vs Lua:

- NPL provides its own set of libraries written in NPL scripts, whose syntax is 100% compatible with Lua. This means that one can use any existing Lua code in NPL script, but the reverse is NOT possible.
- NPL runtime has over 700 000 lines of C/C++ NPL Host API code (not counting third-party library files), on which all NPL library files depends on. In NPL, we advise writing NPL modules in our recommend way, rather than using Lua's `require` function to load any external third-party modules.
- NPL library has over 1 000 000 lines of NPL script code covering many aspects of general-purpose programming. They are distributed as open source NPL packages. You are welcome to contribute your own NPL package utilizing our rich set of host API and libraries.
- NPL provides both non-preemptive and preemptive programming paradigm that is not available in Lua.
- NPL has built-in support for multi-threading and message passing system that is not available in lua.
- Finally, NPL loves Lua syntax. By default all NPL script files has `.lua` extension for ease of editing. Whenever possible, NPL will not deviate its syntax from lua. However, in case, NPL extends Lua syntax, NPL script file will use a different file extension. For example, `*.page`, `*.npl` are used just in case the script uses Non-Lua compatible syntax.

NPL is not a wrapper of lua. It hosts lua's JIT compiler to provide its own programming API and language features, in addition to several other compilers, like mono C#, see below.

Compilers in NPL

NPL runtime state can run on top of three compiler frameworks simultaneously:

- Lua/Luajit compiler
- C/C++
- Mono/C# compiler Which means that you can write code with all NPL API and framework features in either of the above three compilers, and these code can communicate with one another with `NPL.activate` in the same NPL runtime process. Please see NPL Architecture for details.

  Code targeting C/C++ should be pre-compiled. The other two can be compiled on the fly.

References

Figure 1.0: NPL Architecture Stack

CHAPTER 8

NPL Architecture

Figure 1.0: NPL Architecture Stack

Figure 2.0: NPL Scheduler

CHAPTER 9

Getting Started with NPL

The syntax of NPL is 100% compatible with lua. If you are unfamiliar with lua, the only book you need is

Book: Programming in Lua

NPL/Lua Syntax in 15 minutes

The following NPL code is modified from http://learnxinyminutes.com/

```
-- Two dashes start a one-line comment.

--[[
     Adding two ['s and ]'s makes it a multi-line comment.
--]]

----------------------------------------------------
-- 1. Variables and flow control.
----------------------------------------------------

local num = 42;  -- All numbers are doubles.

s = 'string is immutable'
-- comparing two strings is as fast as comparing two pointers
t = "double-quotes are also fine"; -- `;` is optional at line end
u = [[ Double brackets
       start and end
       multi-line strings.]]
-- 'nil' is like null_ptr. It undefines t. When data has no reference,
-- it will be garbage collected some time later.
t = nil

-- Blocks are denoted with keywords like do/end:
while num < 50 do
  num = num + 1  -- No ++ or += type operators.
end

-- If clauses:
if (num <= 40) then
  print('less than 40')
elseif ("string" == 40 or s) then
  print('lua has dynamic type,thus any two objects can be compared.');
elseif s ~= 'NPL' then
```

```
  print('less than 40')
else
  -- Variables are global by default.
  thisIsGlobal = 5  -- variable case is sensitive.

  -- How to make a variable local:
  local line = "this is accessible until next `end` or end of containing script file"

  -- String concatenation uses the .. operator:
  print("First string " .. line)
end

-- Undefined variables return nil.
-- This is not an error:
foo = anUnknownVariable  -- Now foo == nil.

aBoolValue = false

-- Only nil and false are falsy; 0 and '' are true!
if (not aBoolValue) then print('false') end

-- 'or' and 'and' are short-circuited. They are like ||, && in C.
-- This is similar to the a?b:c operator in C:
ans = (aBoolValue and 'yes') or 'no'  --> 'no'

local nSum = 0;
for i = 1, 100 do  -- The range includes both ends. i is a local variable.
  nSum = nSum + i
end

-- Use "100, 1, -1" as the range to count down:
-- In general, the range is begin, end[, step].
nSum = 0
for i = 100, 1, -1 do
   nSum = nSum + i
end

-- Another loop construct:
repeat
  nSum = nSum - 1
until nSum == 0


----------------------------------------------------
-- 2. Functions.
----------------------------------------------------

function fib(n)
  if n < 2 then return 1 end
  return fib(n - 2) + fib(n - 1)
end

-- Closures and anonymous functions are ok:
function adder(x)
  -- The returned function is created when adder is
  -- called, and remembers the value of x:
  return function (y) return x + y end
end
```

```
a1 = adder(9)
a2 = adder(36)
print(a1(16))  --> 25
print(a2(64))  --> 100

-- Returns, func calls, and assignments all work
-- with lists that may be mismatched in length.
-- Unmatched receivers are nil;
-- unmatched senders are discarded.

x, y, z = 1, 2, 3, 4
-- Now x = 1, y = 2, z = 3, and 4 is thrown away.

function bar(a, b, c)
  print(string.format("%s %s %s", a, b or "b", c))
  return 4, 8, 15, 16, 23, 42
end

x, y = bar("NPL")  --> prints "NPL b nil"
-- Now x = 4, y = 8, values 15..42 are discarded.

-- Functions are first-class, may be local/global.
-- These are the same:
function f(x) return x * x end
f = function (x) return x * x end

-- And so are these:
local function g(x) return math.sin(x) end
local g; g  = function (x) return math.sin(x) end

-- Calls with one parameter don't need parenthesis:
print 'hello'  -- Works fine.


----------------------------------------------------
-- 3. Tables.
----------------------------------------------------

-- Tables = Lua's only compound data structure;
--          they are associative arrays.
-- Similar to php arrays or js objects, they are
-- hash-lookup dicts that can also be used as lists.

-- Using tables as dictionaries / maps:

-- Dict literals have string keys by default:
t = {key1 = 'value1', key2 = false}

-- String keys can use js-like dot notation:
print(t.key1)  -- Prints 'value1'.
t.newKey = {}  -- Adds a new key/value pair.
t.key2 = nil   -- Removes key2 from the table.

-- Literal notation for any (non-nil) value as key:
u = {['@!#'] = 'qbert', [{}] = 1729, [6.28] = 'tau'}
print(u[6.28])  -- prints "tau"

-- Key matching is basically by value for numbers
```

```
-- and strings, but by identity for tables.
a = u['@!#']  -- Now a = 'qbert'.
b = u[{}]     -- We might expect 1729, but it's nil:
-- b = nil since the lookup fails. It fails
-- because the key we used is not the same object
-- as the one used to store the original value. So
-- strings & numbers are more portable keys.

-- A one-table-param function call needs no parens:
function h(x) print(x.key1) end
h{key1 = 'Sonmi~451'}  -- Prints 'Sonmi~451'.

for key, val in pairs(u) do  -- Table iteration.
  print(key, val)
end

-- _G is a special table of all globals.
print(_G['_G'] == _G)  -- Prints 'true'.

-- Using tables as lists / arrays:

-- List literals implicitly set up int keys:
v = {'value1', 'value2', 1.21, 'gigawatts'}
for i = 1, #v do  -- #v is the size of v for lists.
  print(v[i])  -- Indices start at 1 !! SO CRAZY!
end
-- A 'list' is not a real type. v is just a table
-- with consecutive integer keys, treated as a list.

----------------------------------------------------
-- 3.1 Metatables and metamethods.
----------------------------------------------------

-- A table can have a metatable that gives the table
-- operator-overloadish behavior. Later we'll see
-- how metatables support js-prototypey behavior.

f1 = {a = 1, b = 2}  -- Represents the fraction a/b.
f2 = {a = 2, b = 3}

-- This would fail:
-- s = f1 + f2

metafraction = {}
function metafraction.__add(f1, f2)
  sum = {}
  sum.b = f1.b * f2.b
  sum.a = f1.a * f2.b + f2.a * f1.b
  return sum
end

setmetatable(f1, metafraction)
setmetatable(f2, metafraction)

s = f1 + f2  -- call __add(f1, f2) on f1's metatable

-- f1, f2 have no key for their metatable, unlike
-- prototypes in js, so you must retrieve it as in
```

```
-- getmetatable(f1). The metatable is a normal table
-- with keys that Lua knows about, like __add.

-- But the next line fails since s has no metatable:
-- t = s + s
-- Class-like patterns given below would fix this.

-- An __index on a metatable overloads dot lookups:
defaultFavs = {animal = 'gru', food = 'donuts'}
myFavs = {food = 'pizza'}
setmetatable(myFavs, {__index = defaultFavs})
eatenBy = myFavs.animal  -- works! thanks, metatable

-- Direct table lookups that fail will retry using
-- the metatable's __index value, and this recurses.

-- An __index value can also be a function(tbl, key)
-- for more customized lookups.

-- Values of __index,add, .. are called metamethods.
-- Full list. Here a is a table with the metamethod.

-- __add(a, b)                     for a + b
-- __sub(a, b)                     for a - b
-- __mul(a, b)                     for a * b
-- __div(a, b)                     for a / b
-- __mod(a, b)                     for a % b
-- __pow(a, b)                     for a ^ b
-- __unm(a)                        for -a
-- __concat(a, b)                  for a .. b
-- __len(a)                        for #a
-- __eq(a, b)                      for a == b
-- __lt(a, b)                      for a < b
-- __le(a, b)                      for a <= b
-- __index(a, b)  <fn or a table>  for a.b
-- __newindex(a, b, c)             for a.b = c
-- __call(a, ...)                  for a(...)

----------------------------------------------------
-- 3.2 Class-like tables and inheritance.
----------------------------------------------------

-- Classes aren't built in; there are different ways
-- to make them using tables and metatables.

-- Explanation for this example is below it.

Dog = {}                                    -- 1.

function Dog:new()                          -- 2.
  newObj = {sound = 'woof'}                 -- 3.
  self.__index = self                       -- 4.
  return setmetatable(newObj, self)         -- 5.
end

function Dog:makeSound()                    -- 6.
  print('I say ' .. self.sound)
end
```

```
mrDog = Dog:new()                                -- 7.
mrDog:makeSound()  -- 'I say woof'               -- 8.

-- 1. Dog acts like a class; it's really a table.
-- 2. function tablename:fn(...) is the same as
--    function tablename.fn(self, ...)
--    The : just adds a first arg called self.
--    Read 7 & 8 below for how self gets its value.
-- 3. newObj will be an instance of class Dog.
-- 4. self = the class being instantiated. Often
--    self = Dog, but inheritance can change it.
--    newObj gets self's functions when we set both
--    newObj's metatable and self's __index to self.
-- 5. Reminder: setmetatable returns its first arg.
-- 6. The : works as in 2, but this time we expect
--    self to be an instance instead of a class.
-- 7. Same as Dog.new(Dog), so self = Dog in new().
-- 8. Same as mrDog.makeSound(mrDog); self = mrDog.

----------------------------------------------------

-- Inheritance example:

LoudDog = Dog:new()                              -- 1.

function LoudDog:makeSound()
  s = self.sound .. ' '                          -- 2.
  print(s .. s .. s)
end

seymour = LoudDog:new()                          -- 3.
seymour:makeSound()  -- 'woof woof woof'         -- 4.

-- 1. LoudDog gets Dog's methods and variables.
-- 2. self has a 'sound' key from new(), see 3.
-- 3. Same as LoudDog.new(LoudDog), and converted to
--    Dog.new(LoudDog) as LoudDog has no 'new' key,
--    but does have __index = Dog on its metatable.
--    Result: seymour's metatable is LoudDog, and
--    LoudDog.__index = LoudDog. So seymour.key will
--    = seymour.key, LoudDog.key, Dog.key, whichever
--    table is the first with the given key.
-- 4. The 'makeSound' key is found in LoudDog; this
--    is the same as LoudDog.makeSound(seymour).

-- If needed, a subclass's new() is like the base's:
function LoudDog:new()
  newObj = {}
  -- set up newObj
  self.__index = self
  return setmetatable(newObj, self)
end

-- Copyright: modified from http://learnxinyminutes.com/
-- Have fun with NPL/Lua!
```

How To Practice

We have developed NPL Code Wiki to help you learn and practice NPL. To launch NPL Code Wiki, you need to

- install Paracraft
- create an empty world or load online world like this one (HourOfCode)
- press F11 to launch NPL Code Wiki.

You can then enter any NPL code in the console. See below:

Advanced Language Features

Lua/NPL is easy to learn, but hard to master. Pay attention to following concepts.

- Lua string is immutable, so comparing two strings is as fast as comparing two pointers.
- Lua has lexical scoping, use it to accelerate your code by exposing frequently used objects as local variables in its upper lexical scope.
- Always use local variables.
- NPL recommends object oriented programming, one can implement all C++ class features with lua tables, like class inheritance and virtual functions, etc.

CHAPTER 10

Hello World in NPL

This tutorial demonstrates

- how to setup a simple development environment
- write a program that output hello world
- explains the basic concept behind the scene

  This article is an aggregates of information you will find in this wiki guide. Skip this tutorial, if you have already read all related pages.

Setup Development Environment

For more information, see Install Guide.

- Download ParacraftSDK
  - or you can clone it by running:

```
git clone https://github.com/LiXizhi/ParaCraftSDK.git
```

- Make sure you have updated to the latest version, by running `./redist/paracraft.exe` at least once
- Run `./NPLRuntime/install.bat`
  - The NPL Runtime will be copied from `./redist` folder to `./NPLRuntime/win/bin`
  - The above path will be added to your %path% variable, so that you can run npl script from any folder.

To write NPL code, we recommend using either or both of following editors (and plugins):

- Visual Studio Community Edition: the free version of `visual studio`
  - In visual studio, select `menu::Tools::Extensions`, and search online for `npl` to install following plugins:
- Visual Studio Code: a cross-platform free editor based on Atom

Hello World Program

Run in command line `npl helloworld.lua`. The content of `helloworld.lua` is like this:

```
print("hello world");
```

You should see a file `./log.txt` in your current working directory, which contains the "hello world" text.

> If `npl` command is not found, make sure `./NPLRuntime/win/bin` is added to your search %path%; or you can simply use the full path to run your script (see below).

```
__YourSDKPath__/NPLRuntime/win/bin/npl helloworld.lua
```

Behind the Scenes

ParacraftSDK's `redist` folder contains an installer of Paracraft. Running `Paracraft.exe` from redist directory will install the latest version of Paracraft, which ships with NPLRuntime. `./NPLRuntime/install.bat` copies only NPL runtime related files from redist folder to `./NPLRuntime/win/bin` and `./NPLRuntime/win/packages`

`./NPLRuntime/win/packages` contains pre-compiled source code of NPL libraries.

To deploy your application, you need to deploy both `bin` and `packages` folders together with your own script and resource files, such as

```
./bin/       :npl runtime 32bits
./bin64/     :npl runtime 64bits
./packages/  :npl precompiled source code
./script/    :your own script
./           :the working (root) directory of your application
```

In NPL script, you can reference any file using path relative to the working directory. So you need to make sure you start your app from the root directory.

Further Readings

ParacraftSDK contains many example projects written in NPL.

- See ParacraftSDK wiki site for more details.
- See Source Code Overview

CHAPTER 11

Video Tutorial: Create A Web Site

video

In this video tutorial, you will learn how to create a web site like this with NPL. The following topics will be covered:

- setup NPL web development environment
- NPL server page programming model
- Debugging and logs
- Use table database
- MVC frameworks for serious web development

  For complete explanation behind the scene, please read WebServer

Setup NPL web development environment

For more information, see Install Guide. The following is only about setting up the recommended IDE under win32.

- Install Git so that you can call `git` command from your cmd line.
- Download ParacraftSDK
  - or you can clone it by running:

```
git clone https://github.com/LiXizhi/ParaCraftSDK.git
```

- Make sure you have updated to the latest version, by running `./redist/paracraft.exe` at least once
- Run `./NPLRuntime/install.bat`
  - The NPL Runtime will be automatically copied from `./redist` folder to `./NPLRuntime/win/bin`
  - The above path will be added to your `%path%` variable, so that you can run npl script from any folder.
- Download and install Visual Studio Community Edition: the free version of `visual studio`
  - In visual studio, select `menu::Tools::Extensions`, and search online for `npl` or `lua` to install following plugins: NPL/Lua language service for visual studio: this will give you all the syntax highlighting, intellisense and debugging capability for NPL in visual studio.

Create a New Visual Studio Project

NPL is a dynamic language, it does not require you to build anything in order to run, so you can create any type of visual studio project you like and begin adding files to it.

There are two ways to do it. Please use `the lazy way` but read `the manual one`.

The lazy way

- run `git clone https://github.com/tatfook/wikicraft.git`
- cd into the source directory and run `./update_packages.bat`
- open the `sln` file with visual studio
- run `start.bat`, you will see the npl server started.
  - visit http://localhost:8099 for the main site.
  - visit http://127.0.0.1:8099 for the NPL code wiki web site.
- rename the project to your preferred name and begin coding in the `./www` folder. (see next chapter)
- finally, see the `configure the project property` step in `the manual one` section.

The manual one

- create an empty visual studio project: such as a C# library or VC++ empty project.
- install the main npl package into your project root directory.

```
mkdir npl_packages
cd npl_packages
git clone https://github.com/NPLPackages/main.git
```

- configure the project property to tell `visual studio` to start NPLRuntime with proper command line parameters when we press `Ctrl+F5`.
  - For the external program: we can use either `npl.bat` or `ParaEngineClient.exe`, see below
  - external program: `D:\lxzsrc\ParaCraftSDKGit\NPLRuntime\win\bin\ParaEngineClient.exe`
  - command line parameters: `bootstrapper="script/apps/WebServer/WebServer.lua" port="8099" root="www/" servermode="true" dev="D:/lxzsrc/wikicraft/"`
  - working directory: `D:/lxzsrc/wikicraft/`
    - Note: the `dev` param and working directory should be the root of your web project directory.
- add main project at `npl_packages/main/*.csproj` to your newly created solution file, so that you have all the source code of NPL and NPL web server for debugging and documentation. **Reopen your visual studio solution file for NPL documentation intellisense to take effect.**
- create `www/` directory and add `index.page` file to it.
- Press `Ctrl+F5` in visual studio, and you will launch the web server.
  - visit http://localhost:8099/index.page in your browser.

NPL server page programming model

NPL web server itself is an open source application written completely in NPL. For its complete source code, please see `script/apps/WebServer/WebServer.lua` in the main npl_package.

So to start a website in a given folder such as `./www/`, we need to call

```
npl bootstrapper="script/apps/WebServer/WebServer.lua" port="8099" root="www/"
```

Normally, we also need to place a file called `webserver.config.xml` in the web root folder. If no config file is found, `default.webserver.config.xml` is used, which will redirect all request without extension to `index.page` and serve `*.npl`, `*.page`, and all static files in the root directory and it will also host NPL code wiki at http://127.0.0.1:8099. For more information, please see WebServer

Play with Mixed Code Web Programming

NPL server page is a mixed HTML/NPL file, usually with the extension `.page`. You can now play with it by creating new `*.page` file in `./www/` directory under visual studio. For more information, see NPLServerPage.

Intellisense in visual studio

If you have installed NPL language service plugin and added main package csproj to your solution, you should already noticed intellisense in your visual studio IDE for all `*.lua,*.npl,*.page` files. Everything under `./Documentation` folder is used for auto-code completion. Please see `npl_packages/main/Documentation/`, you can also add your own intellisense by adding xml files to your own project's './Documentation' folder. For more information, please see NplVisualStudioIDE.

Toggle Page Editing Mode

- One can toggle between two editing mode when writing `*.page` file in visual studio
  - right click the page file in solution manager, and select `open with...`
  - the default text editor will highlight NPL code in the page file, while all HTML text are greyed out
  - the HTML editor will highlight all html tags and all npl code are greyed out. (see below)

You can toggle between these mode depending on whether you are working on HTML or code.

How Does It Work?

At runtime, server page is preprocessed into pure NPL script and then executed.

For example

```
<html><body>
<?npl  for i=1,5 do ?>
   <p>hello</p>
<?npl  end ?>
</body></html>
```

Above server page will be pre-processed into following NPL page script, and cached for subsequent requests.

```
echo ("<html><body>");
for i=1,5 do
 echo("<p>hello</p>")
end
echo ("</body></html>");
```

When running above page script, `echo` command will generate the final HTML response text to be sent back to client.

Internals and Performance

Normally, a single NPL server can serve over 1000 dynamic page requests per seconds and with millions of concurrent connections (limited by your OS). It can serve over 40000 req/sec if it is very simple dynamic page without database query. See NPLPerformance and WebServer for details.

Unlike other preprocessed page language like `PHP`, database queries inside npl page file are actually invoked asynchronously, but written in the more human-friendly synchronous fashion. Please see `Use table database` chapter for details.

Debugging and logs

Most page errors are rendered to the HTTP stream, which is usually directly visible in your browser. For complete or more hidden errors, please keep `./log.txt` in your working directory open, so that you can read most recent error logs in it. Your error is usually either a `syntax` error or a `runtime` error.

To debug with NPL Code wiki

- Launch your website
- Right click on a line in visual studio and select `NPL set breakpoint here` in the context menu.
- The `http://127.0.0.1:8099/debugger` page will be opened.
  - Please note, your actual web site is at `http://localhost:8099`. They are two different websites hosted on the same port by the same NPL process, you can use the first `127.0.0.1` website to debug the second `localhost` website, because both web sites share the same NPL runtime stack/memory/code.

Please see DebugAndLog for general purpose NPL debugging.

Use table database

Table database is written in NPL and is the default and recommended database engine for your website. It can be used without any external dependency or configuration.

```
<html>
<body>
<?
-- connect to TableDatabase (a NoSQL db engine written in NPL)
db = TableDatabase:new():connect("database/npl/", function() end);

db.TestUser:findOne({name="user1"}, resume);
local err, user = yield(); -- async wait when job is done

if(not user) then
    -- insert 5 records to database asynchronously.
```

```
    local finishedCount = 0;
    for i=1, 5 do
        db.TestUser:insertOne({name=("user"..i)}, {name=("user"..i), password="1"},␣
↪function(err, data)
            finishedCount = finishedCount + 1;
            if(finishedCount == 5) then
                resume();
            end
        end);
    end
    yield(); -- async wait when job is done

    echo("<p>we just created "..finishedCount.." users in `./database/npl/TestUser.db`
↪</p>")
else
    echo("<p>users already exist in database, print them</p>")
end

-- fetch all users from database asynchronously.
db.TestUser:find({}, function(err, users)  resume(err, users); end);
err, users = yield(); -- async wait when job is done
?>

<?npl for i, user in ipairs(users) do ?>
    i = <?=i?>, name=<? echo(user.name) ?> <br/>
<?npl end ?>

</body>
</html>
```

Try above code. See UsingTableDatabase for details.

MVC frameworks for serious web development

There are many ways to write MVC framework with NPL. For robust web development, we will introduce the MVC coding style used in this sample web site.

The pros of using MVC framework is that:

- Making your website robust and secure
- Reuse the same code for both client/server side API calls.
- Writing less code for robust http `rest API`

It begins with a router page

All dynamic page requests are handled by a single file `index.page`, which calls `routes.page`.

The router page will check whether the request is a ajax rest API request or a normal page request according to its url. For example, every request url that begins with `/api/[name]/...` is regarded as an API request, and everything else is a normal page request.

If it is a ajax API request, we will simply route it by loading the corresponding page file in `models/[name].page`. In other words, if you placed a `user.page` file in `models` directory, you have already created a set of ajax HTTP rest API on the url `/api/user/...`.

Normally, all of your model file should inherit from a carefully-written `base` model class. So that you can define a robust data-binding interface to your low-level database system or anything else.

For example, the models/project.page, defines basic CRUD interface to `create/get/update/delete` records in the `project.db` database table. It also defines a custom ajax api at `project:api_getminiproject()` which is automatically mapped to the url `api/project/getminiproject/` in the router page (i.e. every method that begins with `api_` in the model class is a public rest API).

How you write your model class and define the mapping rules are completely up to you. The example website just gives you an example of writing model class.

As a website developer, you can choose to query model data on the client side with HTTP calls, or on the server side by directly calling model class method. Generally speaking, client side frameworks like `angular`, etc prefer everything to be called on the client side. We also favors client side query in most cases unless it is absolutely necessary to do it on the server side, such as content needs to be rendered on the server so that it is search-engine friendly.

Final Word

NPL used to be targeted at `3D/TCP server` development since 2005. NPL for Web development was introduced in early 2016, which is relatively new. So if there are errors or questions, please report on our issues.

Hope you enjoy your NPL programming~

CHAPTER 12

Video Tutorial: Create Paracraft Mod

video

In this video tutorial, you will learn how to create 3D applications with NPL. Make sure you have read our previous video TutorialWebSite. This is also going to be a very long video, be prepared and patient!

3D Graphics Powered By ParaEngine

NPL's graphics API is powered by a built-in 3D computer game engine called ParaEngine, which is written as early as 2005 (NPL was started in 2004). Its development goes side-by-side with NPL in the same repository. If you read the source code of NPLRuntime, you will notice two namespaces called `NPL` and `ParaEngine`.

In the past years, I have made several attempts to make the ParaEngine portion more modular, but even so, I did not separate it from the NPLRuntime code tree, because I want the minimum build of NPLRuntime to support all 3D API, just like JAVA, C# and Chrome/NodeJS does. However, if you build NPLRuntime with no `OpenGL/DirectX` graphics, such as on Linux server, these 3D APIs are still available to NPL script but will not draw anythings to your screen.

Introducing Our 3D World Editor: Paracraft

First of all, creating 3D applications isn't easy in general. Fortunately, NPL offers a free and open source NPL package called Paracraft, which is 3d world editor written in NPL.

Paracraft itself is also a standard alone software product for every one (including kids above 7 years old) to create 3D world, animated characters, movies and codes all in a single application. See http://www.paracraft.cn/ for its official website. Actually there are hundreds of interactive 3D worlds, movies and games created by paracraft users without the need of any third-party software.

Fig. Screen shot of Paracraft

> Note: Yes, the look of it are inspired by a great sandbox game called `minecraft` (I love it very much), but using blocks is just the methodology we adopt for crafting objects almost exclusively. Blocks in paracraft can be resized and turned into animated characters. The block methodology (rather than using triangles) allows everyone (including kids) to create models, animations, and even rigging characters very easily, and it is lots of fun. Remember you can still import polygon models from `MAYA/3dsmax`, etc, if you are that professional.

Let us look at some features of Paracraft.

- Everything is a block, entity or item, a block or entity can contain other items in it.

- Every operation is a command and most of them support undo and redo.
- There are items like scripts, movie blocks, bones, terrain brush, physics blocks, etc.
- Use NPL Code Wiki to view and debug.
- Network enabled, you can host your own private world and website in your world directory.

Creating Paracraft Plugin(Mod)

If you want to control more of the logic and looks of your 3D world, you need to write NPL code as professionally as we wrote paracraft itself. This is what this tutorial is really about.

Why Using Mod?

Because paracraft is also released as an open source NPL package, you can just modify the source code of Paracraft and turn it into anything you like or even write everything from scratch.

However, this is not the recommended way, because paracraft is maintained by us and regularly updated. If you directly modify paracraft without informing us, it may be very difficult to stay up to date with us. Unless you find a bug or wrote a new feature which we can merge into our main branch, you are recommended to use Paracraft Mod to create plugins to extend paracraft or even applications that looks entirely different from it.

Paracraft Mod or plugin system exposes many extension points to various aspects of the system. One can use it to add new blocks, commands, web page in NPL code wiki, or even replace the entire user interface and IO of paracraft.

_Fig. Above Image is an online game created via Paracraft Mod by one of our partners. _

You can see how much difference there is from paracraft.

Creating Your First Paracraft Mod

Install ParacraftSDK

- Install Git so that you can call `git` command from your cmd line.
- Download ParacraftSDK
  - or you can clone it by running:

```
git clone https://github.com/LiXizhi/ParaCraftSDK.git
```

- Make sure you have updated to the latest version, by running `./redist/paracraft.exe` at least once
- Download and install Visual Studio Community Edition: the free version of `visual studio`
  - In visual studio, select `menu::Tools::Extensions`, and search online for `npl` or `lua` to install following plugins: NPL/Lua language service for visual studio: this will give you all the syntax highlighting, intellisense and debugging capability for NPL in visual studio.

see Install Guide and ParaCraftSDK Simple Plugin for more information.

Create `HelloWorld` Mod

- Run `./bin/CreateParacraftMod.bat` and enter the name of your plugin, such as `HelloWorld`. A folder at `./_mod/HelloWorld` will be generated.
- Run `./_mod/HelloWorld/Run.bat` to test your plugin with your copy of paracraft in the ./redist folder.

The empty plugin does not do anything yet, except printing a log when it is loaded.

Command line and runtime environment

If you open Run.bat with a text editor, you will see following lines.

```
@echo off
pushd "%~dp0../../redist/"
call "ParaEngineClient.exe" dev="%~dp0" mod="HelloWorld" isDevEnv="true"
popd
```

It specifies following things:

- the application is started via ParaEngineClient.exe in `./redist` folder
- the `dev` parameter specifies the development directory (which is the plugin directory ./_mod/HelloWorld/), basically what this means is that NPLRuntime will always look for files in this folder first and then in working directory (`./redist` folder).
- the `mod` and `isDevEnv` parameters simply tells paracraft to load the given plugin from the development directory, so that you do not have to load and enable it manually in paracraft GUI.

Install Source Code From NPL Package

All APIs of paracraft are available as source code from NPL package. Run `./_mod/HelloWorld/InstallPackages.bat` to get them or you can create `./_mod/HelloWorld/npl_packages` and install manually like this.

```
cd npl_packages
git clone https://github.com/NPLPackages/main.git
git clone https://github.com/NPLPackages/paracraft.git
```

You may edit `InstallPackages.bat` to install other packages, and run it regularly to stay up-to-date with git source.

> it is NOT advised to modify or add files in the ./npl_packages folder, instead create a similar directory structure in your mod directory if you want to add or modify package source code. Read npl_packages for how to contribute.

Setup Project Files

NPL is a dynamic language, it does not require you to build anything in order to run, so you can create any type of visual studio project you like and begin adding files to it. Open `ModProject.sln` with visual studio in your mod directory, everything should already have been setup for you. You can `Ctrl+F5` to run.

To manually create project solution file, follow following steps:

- create an empty visual studio project: such as a C# library and remove all cs files from it.

- add `npl_packages/main/*.csproj` and `npl_packages/paracraft/*.csproj` to your solutions, so that you have all the source code of NPL core lib and paracraft, where you can easily search for documentation and implementation.
- configure the project property to tell visual studio to start NPLRuntime with proper command line parameters when we press Ctrl+F5. The following does exactly the same thing as the `./Run.bat` in mod folder.
  - For the external program: we can use `ParaEngineClient.exe`, see below
  - external program: `D:\lxzsrc\ParaCraftSDKGit\redist\ParaEngineClient.exe`
  - command line parameters: `mod="HelloWorld" isDevEnv="true" dev="D:/lxzsrc/ParaCraftSDKGit/redist/_mod/HelloWorld/"`
  - working directory: `D:/lxzsrc/ParaCraftSDKGit/redist/_mod/HelloWorld/`
    - Note: the `dev` param and working directory should be the root of your mod project folder.

Please see my previous video tutorial for web server for more details about installing NPL language service for visual studio.

Understanding Paracraft Mod

When paracraft starts, it will load the file at `./Mod/_plugin_name_/main.lua` for each enabled plugin. It is the entry point of any plugin, everything else is up to the developer.

In our example, open the entry file `./_mod/HelloWorld/Mod/HelloWorld/main.lua` (see below). Because development directory is set to `_mod/HelloWorld` by startup command line, the relative path of above file is `./Mod/HelloWorld/main.lua`. This is why Paracraft can find the entry file.

```
local HelloWorld = commonlib.inherit(commonlib.gettable("Mod.ModBase"),commonlib.
↪gettable("Mod.HelloWorld"));

function HelloWorld:ctor()
end

-- virtual function get mod name
function HelloWorld:GetName()
    return "HelloWorld"
end

-- virtual function get mod description
function HelloWorld:GetDesc()
    return "HelloWorld is a plugin in paracraft"
end

function HelloWorld:init()
    LOG.std(nil, "info", "HelloWorld", "plugin initialized");
end

function HelloWorld:OnLogin()
end

-- called when a new world is loaded.
function HelloWorld:OnWorldLoad()
end

-- called when a world is unloaded.
function HelloWorld:OnLeaveWorld()
end
```

```
function HelloWorld:OnDestroy()
end
```

Most plugin will register a class in the "Mod" table, like above. The class usually inherits from `Mod.ModBase`. The class will be instantiated when paracraft starts, and its virtual functions are called at appropriate time.

For example, if your plugin contains custom commands or block items, you can register them in the `init()` method.

How to write plugin

There are three recommended ways to extend paracraft

- create your own command: command is a powerful way to interact with the user. The rule of thumb is that **before you have any GUI dialog, consider writing a command first**. Command is easy to test, integrate, and capable of interacting with other commands. Graphic User Interface(GUI) is good, however, it can only interact with humans in strict ways, commands can interact with both human and other commands or programs.
- create custom items: custom items include entities, items, or blocks. This ensures that your code is modular and independent. The best way to add new functions to paracraft is by adding new block or item types. Consider the color block and movie block in paracraft, which contains both GUI and complex logics.
- add filters: filters is a useful way to inject your code at predefined integration points in Paracraft.

The least recommended way to extend paracraft is to clone the source code files of paracraft to development directory and modify them. Because plugin directory is searched before the main paracraft source code pkg file, it is possible to overwrite source code of paracraft in this way, but you are highly decouraged to do so. If you are requesting a feature that can not be added via the recommended ways, please inform the developers of paracraft to add a new filter for you, instead of modifying our source code directly in your plugin.

Extending Command

see `_mod/Test/Mod/Test/DemoCommand.lua`

Extending GUI

see `_mod/Test/Mod/Test/DemoGUI.lua`

Extending Item

TODO:

Extending NPL Code Wiki

Users of Paracraft knows about `NPL Code Wiki`, if you want to use latest web technology (HTML5/js) to create user interface that can interact with your 3D application, you can extend the NPL code wiki.

Now let us create a new web page and add a menu item in NPL Code wiki

First of all, you need to place your server page file in NPL code wiki's web root directory. So we can create a file at `./_mod/HelloWorld/script/apps/WebServer/admin/wp-content/pages/HelloWorld.page`

its content may be

```
<h1>Hello World from plugin!</h1>
```

In the `./_mod/HelloWorld/Mod/HelloWorld/main.lua` file's init function, we add a menu filter to attach a menu item.

```
function HelloWorld:init()

    -- add a menu item to NPL code wiki's `Tools:helloworld`
    NPL.load("(gl)script/apps/WebServer/WebServer.lua");
    WebServer:GetFilters():add_filter( 'wp_nav_menu_objects', function(sorted_menu_
↪items)
        sorted_menu_items[sorted_menu_items:size()+1] = {
            url="helloworld",
            menu_item_parent="Tools",
            title="hello world",
            id="helloworld",
        };
        return sorted_menu_items;
    end);
end
```

Now restart paracraft, and press F11 to activate the NPL code wiki, you will notice the new menu item in its tools menu.

Documentation

The best documentation is the source code of paracraft which is in the paracraft package and several plugin demos in the `./_mod/test` folder. Since code is fully documented, use search by text in source code for documentations, as if you are googling on the web.

Debugging

For detailed information, please see NPL Debugging and Logs

- Use `./redist/log.txt` to print and check log file.
- press `F12` to bring up the buildin debugger.
- use `Ctrl+F3` to use the MCML browser for building GUI pages.
- use GUI debuggers for NPL in visual studio.

Plugin Deployment

To deploy and test your plugin in production environment, you need to package all the files in your plugin's development directory (i.e. `_mod/HelloWorld/*.*`) into a zip file called `HelloWorld.zip`, and place the zip file to './redist/Mod/HelloWorld.zip'. It is important that the file path in the zip file resembles the relative path to the development directory (i.e. `Mod/HelloWorld/main.lua` in the `HelloWorld.zip`).

Once you have deployed your plugin, you can start paracraft normally, such as running `./redist/Paracraft.exe`, enable your plugin in Paracraft and test it.

You can redistribute your plugins in any way you like. We highly recommend that you code and release on github and inform us (via email or pull request) if you have just released some cool plugin. We are very likely to fork your code and review it.

Further Readings

- Check out all sample plugins in `./_mod` folder.
- Check out all the SDK video tutorials by the author of Paracraft on the website.
- Read NPL Runtime's official wiki

Why are you doing this?

Lots of people and even our partners asked me why are you doing this `NPL/Paracraft/ParaEngine` thing and make all of them open source? To make the long story short, I want to solve two problems:

- Create a self-learning based educational platform so that everyone can learn computer science as my own childhood does.
  - We hope in 7 years, this could be the best present that we can give to our children. Right now, I really cannot find any material even close to it on the Internet.
- Create a online 3D simulated virtual world in which we can test new artificial intelligence algorithms with lots of humans. Using blocks is obviously the most economical way.

Thanks to our generous investor at `Tatfook`, we are able to sponsor more independent programmers and support more teams and partners using NPL to fulfill the above two goals. Please read this if you are interested in joining us and be sponsored.

CHAPTER 13

NPL Source Code Overview

This is a brief overview of NPL source code. The best way to learn NPL is to read its source code.

All NPL related source code is distributed as packages.

Two important packages for NPL and paracraft developement is at:

- main package: NPL standard library source code
- paracraft package: paracraft source code

```
cd npl_packages
git clone https://github.com/NPLPackages/main.git
git clone https://github.com/NPLPackages/paracraft.git
```

When you program, you can search by text in all NPL source code.

Source code overview

Core Library Files

- script/ide/
- script/ide/commonlib.lua: common include file for frequently used libraries without GUI.
- script/ide/System: Core library
- script/ide/STL: Like C++ stl, simple dData structures
- script/ide/timer.lua: time class
- script/ide/math: vector, matrix, etc.

Object Oriented

- script/ide/oo.lua: class inheritance implementation
- script/ide/System/Core/ToolBase.lua: a powerful base class with signal/property support.

Web Server Source Code

- script/apps/WebServer: implementation of a web server (like Apache) in NPL.
- script/apps/WebServer/WebServer.lua: entry file
- script/apps/WebServer/admin: A php-like web site framework in NPL

Paracraft Source Code

Lots of code here. If you are developing Paracraft plugins or 3d client applications in NPL. It is important to read source here.

- script/apps/Aries/Creator/Game: main source code of Paracraft
- [script/apps/Aries/Creator/Game/Areas](https://github.com/NPLPackages/paracraft/tree/master/script/apps/Aries/Creator/Game/Areas): desktop GUI
- script/apps/Aries/Creator/Game/Entity: All movable entity types
- script/apps/Aries/Creator/Game/blocks: All block types
- [script/apps/Aries/Creator/Game/Commands](https://github.com/NPLPackages/paracraft/tree/master/script/apps/Aries/Creator/Game/Commands): In-Game Commands
- [script/apps/Aries/Creator/Game/Tasks](https://github.com/NPLPackages/paracraft/tree/master/script/apps/Aries/Creator/Game/Tasks): Commands with GUI
- [script/apps/Aries/Creator/Game/GUI](https://github.com/NPLPackages/paracraft/tree/master/script/apps/Aries/Creator/Game/GUI): editor GUIs
- [script/apps/Aries/Creator/Game/Network](https://github.com/NPLPackages/paracraft/tree/master/script/apps/Aries/Creator/Game/Network): Paracraft server implementation
- [script/apps/Aries/Creator/Game/Shaders](https://github.com/NPLPackages/paracraft/tree/master/script/apps/Aries/Creator/Game/Shaders): GPU shaders
- [script/apps/Aries/Creator/Game/Mod](https://github.com/NPLPackages/paracraft/tree/master/script/apps/Aries/Creator/Game/Mod): Plugin interface
- script/apps/Aries/Creator/Game/SceneContext: User input like keyboard and mouse

CHAPTER 14

NPL Code Wiki

NPL Code Wiki is a web site which can be served directly from your local computer using NPL Runtime. It provides you a way to communicate with NPL Runtime via any web browser.

- NPL Code Wiki is designed to help people learn and practice NPL programming.
- It provides a way to inspect or debug NPL Runtime via HTTP protocol.
- Providing an interactive console to edit/debug NPL code at runtime.

Start Code Wiki

Start With NPL Runtime

Run from console

```
npl script/apps/WebServer/WebServer.lua
```

The default web address is http://127.0.0.1:8099/. For more information, please see WebServer section.

Start With Paracraft

To launch NPL Code Wiki, you need to

- install Paracraft
- create an empty world
- press F11 to launch. NPL Code wiki will be opened using your default web browser.

You can then enter any NPL code in the console. See below:

NPL Code Wiki Source Code is at script/apps/WebServer

Alternatively, you can start it via following code. Press `F12` in any world in Paracraft to enter code.

```
NPL.load("(gl)script/apps/WebServer/WebServer.lua");
WebServer:Start("script/apps/WebServer/admin");
```

NPL Debugger

NPL debugger is a HTTP protocol based debugger. It can be opened from `menu::tools::NPL debugger`, or with `Ctrl+Alt+I` in paracraft.

Because it is based on HTTP and served from the same NPL runtime you are debugging, you can debug your running application from any modern web browser. For example, you can debug apps running on linux server or mobile device remotely from a windows client. However, you must make sure that the apps are started with full source code available in its running directory.

How to Attach

There is no need to start your app in debug mode. Instead, just launch NPL code wiki, and follow the steps:

- In the web page, open your script files and press F9 or click the left margin of the line to set breakpoints.
- Click `Attach` button to begin debugging.
  - You can then press F10, F11, Shift+F11 or press the buttons to step through your code.
  - In the watcher window, you can evaluate any code or variable. Just type the variable name or code and click `evaluate`
- When debugger is attached, your program may run pretty slow, so once you are finished, click `Stop` button to detach.

Limitations

Currently only the main thread can be debugged, if you want to debug other thread, you can write your own debugger in NPL, the complete source code of the NPL debugger web site is only 400 lines.

NPL Code Editor

This is a multi-tab code editor for NPL. It can be opened from menu::view::code editor.

- Enter file name and press enter to open the file.
- Click the `sync` button on left top corner to navigate to the active document's directory.
- Even if you close your computer, your last opened files will be remembered when you open it again.
- Right click on the file name tab for a list of commands, such as `open containing folder`, `reload`, etc.
- Modified files will be marked with `*` in its tab, and need to be confirmed before closing it.
- `Ctrl+S` to save to disk.
- Filter files as you type in the left top's directory text box.

Object Inspector

Click `Menu::View::ObjectInspect` to open `object inspector`. Here you can view and edit all objects in NPL Runtime.

Once you edited a property, a command string is generated on top of the property window. The command string is equivalent to your last operation. In above image, the command string to change the window title is:

```
/property -all WindowText Paracraft
```

> The server-side NPL code that realized the `Object Inspector` page can be viewed by clicking the `view source` link on the bottom of the web page. This provides you a way to study the NPL implementation of whatever you see in NPL Code wiki web site.

The Console Page

Click `Menu::Tools::Console` to open `Console Window`. Here you can enter any NPL code or NPL web page code to be evalulated at runtime.

For example, enter following code and press `Run as Code` button:

```
print("hello world")
```

By default, text is output to `log.txt` file in the current working directory.

```
alert("hello world")
```

The above code will display a message box, if code wiki is launched with Paracraft.

The Log Page

Click `Menu::Tools::Log` to open `Log Window`. Both the console page and the log page can display content in `log.txt`.

- One can leave this window open even when your application restarts. It will automatically scroll to the newest log for you.
- Check `preserve log` to preserve logs between each application restarts.
- If there are critical errors or warnings, an error sign with a text number will be displayed in red on top of the log window, click it to navigate to the line of interest in the log file.

CHAPTER 15

Debugging And Logs

There are many ways to debug your NPL scripts, including printing logs, via NPL Code Wiki, or via IDE like visual studio or vscode, etc.

Logs in NPL

`log.txt` file is the most important place to look for errors. When your application starts, NPL runtime will create a new `log.txt` in the current working directory.

Rule of Silence

The entire Unix system is built on top of the philosophy of rule of silence, which advocate every program to output nothing to standard output unless absolutely necessary.

So by default, NPL runtime output nothing to its standard output. All NPL output API like `log`, `print`, `echo` write content to `log.txt` instead of standard output. There is only one hidden API `io.write` which can be used to write to standard output.

By default, NPL runtime and its libraries output many info to log files. This log file can grow big if your server runs for days. There are two ways to deal with it: one is to use a timer (or external shell script) to backup or redirect `log.txt` to a different file everyday; second is to change the default log level to `error`, which only output fatal errors to `log.txt`. Application developers can also write their own logs to several files simultaneously, please see `script/ide/log.lua` for details.

When application starts, one can also change the default log file name, or whether it will overwrite existing `log.txt` or append to it. Please refer to log interface C++ API for details.

Use Logs For Debugging

The default log level of NPL is `TRACE`, which will write lots of logs to `log.txt`. At development time, it is good practice to keep this file open in a text editor that can automatically refresh when file content is changed.

When you feel something goes wrong, search for `error` or `runtime error` in `log.txt`. NPL Code Wiki provide a log page from `menu:view:log`, where you can filter logs. See below:

Many script programmers prefer to insert `echo({params,...})` in code to output debug info to `log.txt`, and when code is working, these `echo` lines are commented out.

A good practice is to insert permanent log line with debug level set to `debug` like `LOG.std(nil,"debug","modulename","some text",...)`.

Use A Graphical Debugger

HTTP Debugger in NPL Code Wiki

Sometimes inserting temporary log lines into code can be inefficient. One can also use NPL HTTP debugger to debug code via HTTP in a HTML5 web browser with nice GUI. NPL HTTP Debugger allows you to debug remote process, such as debugging linux/android/ios NPL process on a windows or mac computer. See NPLCodeWiki for details.

If you have installed NPL/Lua language service for visual studio, one can set HTTP-based breakpoints directly inside visual studio.

In visual studio, open your NPL script or page file, right click the line where you want to set breakpoint, and from the context menu, select `NPL Set Breakpoint Here`.

If you have started your NPL process, visual studio will open the HTTP debugger page with your default web browser. This is usually something like `http://localhost:8099/debugger`. Breakpoints will be automatically set and file opened in the browser.

Debugger Plugin for Visual Studio

We also provide open source NPL debugger plugins that is integrated with visual studio and vscode, see InstallGuide for details.However, it is a little complex to install visual studio debugger plugin, please follow instruction here: How To Install NPL Debugger for VS

Once installed, you can debug by selecting from visual studio menu `Tools::Launch NPL Debugger...`

CHAPTER 16

Embedding NPLRuntime

NPLRuntime comes with a general purpose executable for launching NPL based applications. This is the standard usage of NPLRuntime.

However, some client application may prefer embedding NPLRuntime in their own executable file, or even as child window of their own application. This is possible via NPLRuntime dll or (`paraengineclient.dll`) under windows platform.

Please see https://github.com/LiXizhi/ParaCraftSDK/blob/master/samples/MyAppExe/MyApp.cpp for details.

```
#include "PEtypes.h"
#include "IParaEngineApp.h"
#include "IParaEngineCore.h"
#include "INPL.h"
#include "INPLRuntime.h"
#include "INPLRuntimeState.h"
#include "INPLAcitvationFile.h"
#include "NPLInterface.hpp"
#include "PluginLoader.hpp"

#include "MyApp.h"

using namespace ParaEngine;
using namespace NPL;
using namespace MyCompany;

// some lines omitted here ....

INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR lpCmdLine, INT)
{
    std::string sAppCmdLine;
    if (lpCmdLine)
        sAppCmdLine = lpCmdLine;

    // TODO: add your custom command line here
    sAppCmdLine += " mc=true noupdate=true";

    CMyApp myApp(hInst);
    return myApp.Run(0, sAppCmdLine.c_str());
}
```

Building Custom Executable Application

To build your custom application, you must download NPLRuntime source code (for header files) and set CMAKE include directory properly.

- Download the sample at https://github.com/LiXizhi/ParaCraftSDK/blob/master/samples/MyAppExe
- Set CMAKE include directory to https://github.com/LiXizhi/NPLRuntime/tree/master/Client/trunk/ParaEngineClient/Core
- `paraengineclient.dll` is dynamically loaded at runtime via NPL plugin interface, which means that your executable does not need to link to any library file. **All you need when embedding NPLRuntime is header files. ** This decouples your host application executable from NPLRuntime library files which may be upgraded separately without rebuilding your host executable.
  - Main header files:

Build NPLRuntime as Libraries

In most cases, you do not need to build these libraries, simply copy `ParaEngineClient.dll` and other related dll files from the ParacraftSDK's `./redist` folder to your host application's executable directory. Make sure they are up to date.

However, one can also build NPL Runtime as Libraries manually. See `CMake` options when building NPLRuntime from source code.

CHAPTER 17

Extending NPLRuntime With C++ Plugins

NPL provides three extensibility modes: (1) NPL scripting (2) Mono C# dll (3) C++ plugin interface. All of them can be used simultaneously. Please see BasicConcept for details.

C++ plugins

C++ plugins allows us to treat dll/so file as a single script file. We would only want to use it for performance critical tasks or functions that make heavy use of other third-party C/C++ libraries.

The following is a sample c++ plugin. https://github.com/LiXizhi/ParaCraftSDK/tree/master/samples/plugins/HelloWorldCppDll

```
#include "stdafx.h"
#include "HelloWorld.h"

#ifdef WIN32
#define CORE_EXPORT_DECL    __declspec(dllexport)
#else
#define CORE_EXPORT_DECL
#endif

// forward declare of exported functions.
#ifdef __cplusplus
extern "C" {
#endif
    CORE_EXPORT_DECL const char* LibDescription();
    CORE_EXPORT_DECL int LibNumberClasses();
    CORE_EXPORT_DECL unsigned long LibVersion();
    CORE_EXPORT_DECL ParaEngine::ClassDescriptor* LibClassDesc(int i);
    CORE_EXPORT_DECL void LibInit();
    CORE_EXPORT_DECL void LibActivate(int nType, void* pVoid);
#ifdef __cplusplus
}   /* extern "C" */
#endif

HINSTANCE Instance = NULL;

using namespace ParaEngine;

ClassDescriptor* HelloWorldPlugin_GetClassDesc();
typedef ClassDescriptor* (*GetClassDescMethod)();
```

```
GetClassDescMethod Plugins[] =
{
    HelloWorldPlugin_GetClassDesc,
};

/** This has to be unique, change this id for each new plugin.
*/
#define HelloWorld_CLASS_ID Class_ID(0x2b905a29, 0x47b409af)

class HelloWorldPluginDesc:public ClassDescriptor
{
public:
    void* Create(bool loading = FALSE)
    {
        return new CHelloWorld();
    }

    const char* ClassName()
    {
        return "IHelloWorld";
    }

    SClass_ID SuperClassID()
    {
        return OBJECT_MODIFIER_CLASS_ID;
    }

    Class_ID ClassID()
    {
        return HelloWorld_CLASS_ID;
    }

    const char* Category()
    {
        return "HelloWorld";
    }

    const char* InternalName()
    {
        return "HelloWorld";
    }

    HINSTANCE HInstance()
    {
        extern HINSTANCE Instance;
        return Instance;
    }
};

ClassDescriptor* HelloWorldPlugin_GetClassDesc()
{
    static HelloWorldPluginDesc s_desc;
    return &s_desc;
}

CORE_EXPORT_DECL const char* LibDescription()
{
    return "ParaEngine HelloWorld Ver 1.0.0";
```

```
}

CORE_EXPORT_DECL unsigned long LibVersion()
{
    return 1;
}

CORE_EXPORT_DECL int LibNumberClasses()
{
    return sizeof(Plugins)/sizeof(Plugins[0]);
}

CORE_EXPORT_DECL ClassDescriptor* LibClassDesc(int i)
{
    if (i < LibNumberClasses() && Plugins[i])
    {
        return Plugins[i]();
    }
    else
    {
        return NULL;
    }
}

CORE_EXPORT_DECL void LibInit()
{
}

#ifdef WIN32
BOOL WINAPI DllMain(HINSTANCE hinstDLL,ULONG fdwReason,LPVOID lpvReserved)
#else
void __attribute__ ((constructor)) DllMain()
#endif
{
    // TODO: dll start up code here
#ifdef WIN32
    Instance = hinstDLL;                // Hang on to this DLL's instance handle.
    return (TRUE);
#endif
}

/** this is the main activate function to be called, when someone calls NPL.activate(
↪"this_file.dll", msg);
*/
CORE_EXPORT_DECL void LibActivate(int nType, void* pVoid)
{
    if(nType == ParaEngine::PluginActType_STATE)
    {
        NPL::INPLRuntimeState* pState = (NPL::INPLRuntimeState*)pVoid;
        const char* sMsg = pState->GetCurrentMsg();
        int nMsgLength = pState->GetCurrentMsgLength();

        NPLInterface::NPLObjectProxy input_msg = NPLInterface::NPLHelper::
↪MsgStringToNPLTable(sMsg);
        const std::string& sCmd = input_msg["cmd"];
        if(sCmd == "hello" || true)
        {
            NPLInterface::NPLObjectProxy output_msg;
```

```
            output_msg["succeed"] = "true";
            output_msg["sample_number_output"] = (double)(1234567);
            output_msg["result"] = "hello world!";

            std::string output;
            NPLInterface::NPLHelper::NPLTableToString("msg", output_msg, output);
            pState->activate("script/test/echo.lua", output.c_str(), output.size());
        }
    }
}
```

The most important function is `LibActivate` which is like `NPL.this(function() end)` in NPL script.

It is the function to be invoked when someone calls something like below:

```
NPL.activate("this_file.dll", {cmd="hello"})
```

One can also place dll or so file under sub directories of the working directory, such as `NPL.activate("mod/this_file.dll",{cmd="hello"})`

Note to Linux Developers

Under linux, the generated plugin file name is usually "libXXX.so", please copy this file to working directory and load using `NPL.activate("XXX.dll",{cmd="hello"})` instead of the actual filename. NPLRuntime will automatically replace `XXX.dll` with `libXXX.so` when searching for the dll file. This way, your script code does not need to change when deploying to linux and windows platform.

CHAPTER 18

NPL Basic Concept

This section provides a more detailed description to NPL's language feature.

A Brief History of Computer Language

Computer programming language is essentially a language in itself, like English and Chinese; but it is mainly written by humans to instruct computers. Our current computers has its unique way to process language. In general, it is a kind of command-like IO system, with input and output. Any piece of code can always be statically mapped to nodes and connections, where each connection has a single-unique direction from output to input; when a computer executes the code, it moves data along those connections. This is similar to how our brain works by our biological neural network. The difference is that we do not know how, when and where data is passed along. Since parallelism in our brain can also be mapped to a certain kind of node-connection relationship; the only thing differs is computational speed. So when designing computer language, let us first take concurrent programming out, and focus on making it easier and efficient to create all kinds of nodes and connections, and passing data along the connections.

Let us list all major patterns:

- Nearly all languages provides functions:
  - Functional language like `C` provides defining custom input/output patterns with functions. Let us regard `if`,`else`,`expressions`, etc as build-in functions.
- In some languages like `javascript`, function is first-class object, allowing one to create interesting node-connection patterns.
- Some like `C++`,`java`,`php` supports object-oriented programming, which is essentially a filter-like input/output patterns, that moves data as a whole more easily.
- Some is weakly typed like `javascript`, `python` allowing a node to accept input from more versatile input, at some expense. Some strongly-typed language like `C++` use template-programming to simulate this feature at the cost of more code generated.
- Some is statically scoped like `lua`, that automatically captured variables for any asynchronous callback, and some dynamically scoped language also simulates this feature with anonymous function and captures in some clumsy way like 'C#'.
- Some features establishing node and connections across multiple threads or computers on the network.

So what is the language that support all the above, while still maintaining high-performance? A close answer is `lua`; the complete answer is `NPL`.

What NPL is Not

- `meta programming` is a technique to translate text code from first domain to second one, and then compile second one and execute. It is common for some library to use this technique to provide a more concise way of coding with the library. Examples:
  - `qt.io` in C++
  - `php` which pre-process code-mixed hyper text into complete programming code.
  - `Scala` to 'Java'
  - `CoffeeScript` to `Javascript`
- NPL is NOT `meta programming`. But NPL web server library use a similar technique to make coding server web pages that generate HTML5/javascript on the client more easily.
- NPL is not syntactic sugar or meta-programming for concurrent programming library. It is the programmer's job to write or use libraries that support concurrent programming, rather than introducing some non-intuitive syntax to the language itself.

NPL Runtime Architecture

This section helps you to understand the internals of NPL runtime. One of the major things that one need to understand is its message passing model.

Message Passing Model

In NPL, any file on disk can receive messages from other files. A file can send messages to other files via `NPL.activate` function. This function is strictly asynchronous. The details is below:

Message processing always has two phases:

- phase1: synapse data relay phase
- phase2: neuron file response phase

When you send a message via `NPL.activate`, the message is in phase 1. Message is not processed immediately, but cached in a queue. For example, if the target neuron file is on the local machine, the message is pushed (cached) directly to local thread queue; if the target is on remote machine, NPL runtime will automatcally establish TCP connection, and it will ensure the message is pushed(cached) on the target machine's specified thread queue.

Message processing takes place in phase2. Each NPL runtime process may contain one or more NPL runtime states, each corresponds to one system-level thread with its own message queue, memory, and code. NPL runtime states are logically separated from each other. In each NPL state, it processes messages in its message queue in heart-beat fashion. Therefore it is the programmer's job to finish processing the message within reasonable time slice, otherwise the message queue will be full and begin to lose early messages.

Code Overview

This section helps you to understand the open source code of NPL.

incoming connections

```
incoming connection:  acceptor/dispatcher thread
    -> CNPLNetServer::handle_accept
        -> CNPLNetServer.m_connection_manager.start(a new CNPLConnection object) : a␣
↪new CNPLConnection object is added to CNPLConnectionManager
            -> CNPLConnection.start()
                -> automatically assign a temporary nid to this unauthenticated␣
↪connection, temp connection is called ~XXX.
                    -> CNPLDispatcher::AddNPLConnection(temporary nid, CNPLConnection)
                -> begin reading the connection sockets
                -> LATER: Once authenticated, the scripting interface can call␣
↪(CNPLConnection)NPL.GetConnection(nid)::rename(nid, bAuthenticated)
```

if any messages are sent via the new incomming connection, the message field should contain msg.nid and msg.tid, where tid is temporaray nid. If there is no nid, it means that the connection is not yet authenticated. In NPL, one can call `NPL.accept(tid,nid)`

incoming messages

```
incoming message
    -> CNPLConnection::handle_read
        -> CNPLConnection::handleReceivedData
            -> CNPLConnection::m_parser.parse until a complete message is read
                -> CNPLConnection::handleMessageIn()
                    -> CNPLConnection::m_msg_dispatcher.DispatchMsg((NPLMsgIn)this->m_
↪input_msg)
                        -> CNPLRuntimeState = GetNPLRuntime()->
↪GetRuntimeState(NPLMsgIn::m_rts_name)
                        -> if CNPLDispatcher::CheckPubFile(NPLMsgIn::m_filename)
                            -> new NPLMessage(from NPLMsgIn)
                            -> CNPLRuntimeState::Activate_async(NPLMessage);
                                -> CNPLRuntimeState::SendMessage(NPLMessage) ->␣
↪insert message to CNPLRuntimeState.m_input_queue
                        -> or return access denied
```

outgoing messages

```
from NPL script NPL.activate()
    -> CNPLRuntime::NPL_Activate
        -> if local activation, get the runtime state.
            -> CNPLRuntimeState::Activate_async(NPLMessage) -> CNPLRuntimeState::
↪SendMessage(NPLMessage) -> insert message to CNPLRuntimeState.m_input_queue
        -> if remote activation with nid, send via dispatcher
            -> CNPLDispatcher::Activate_Async(NPLFileName, code)
                -> CNPLConnection = CNPLDispatcher::CreateGetNPLConnectionByNID(nid)
                    -> if found in CNPLDispatcher::m_active_connection_map(nid,␣
↪CNPLConnection), return
                    -> or if found in CNPLDispatcher::m_pending_connection_map(nid,␣
↪CNPLConnection), return
                    -> or if found in CNPLDispatcher::m_server_address_map(host, port,
↪ nid): we will actively establish a new connection to it.
                        -> CNPLDispatcher::CreateConnection(NPLRuntimeAddress)
                            -> CNPLNetServer::CreateConnection(NPLRuntimeAddress)
```

```
                                -> new CNPLConnection()
                                -> CNPLConnection::
↪SetNPLRuntimeAddress(NPLRuntimeAddress)
                                -> (CNPLNetServer::m_connection_manager as␣
↪CNPLConnectionManager)::add(CNPLConnection)
                                -> CNPLConnection::connect() -> CNPLConnection::m_
↪resolver->async_resolve ->
                                    -> CNPLConnection::handle_resolve ->␣
↪CNPLConnection::m_socket.async_connect
                                    -> CNPLConnection::handle_connect
                                        -> CNPLConnection::handleConnect() : for␣
↪custom behavior
                                        -> CNPLConnection::start()
                                            -> CNPLDispatcher::
↪AddNPLConnection(CNPLConnection.m_address.nid, CNPLConnection)
                                                -> add nid to CNPLDispatcher::m_
↪active_connection_map(nid, CNPLConnection), remove nid from m_pending_connection_
↪map(nid, CNPLConnection)
                                            -> begin reading the connection sockets
                            -> Add to CNPLDispatcher::m_pending_connection_map if not␣
↪connected when returned, return the newly created connection
                    -> or return null.
                -> CNPLConnection::SendMessage(NPLFilename, code)
                    -> new NPLMsgOut( from NPLFilename, code)
                    -> push to CNPLConnection::m_queueOutput() and start sending␣
↪first one if queue is previously empty.
```

NPL Message Format

NPL message protocol is based on TCP and HTTP compatible. More information is in NPLMsgIn_parser.h

quick sample

```
A (g1)script/hello.lua NPL/1.0
rts:r1
User-Agent:NPL
16:{"hello world!"}
```

NPL Quick Guide

In NPL runtime, there can be one or more runtime threads called NPL runtime states. Each runtime state has its own name, stack, thread local memory manager, and message queue. The default NPL state is called (main), which is also the thread for rendering 3D graphics. It is also the thread to spawn other worker threads.

To make a NPL runtime accessible by other NPL runtimes, one must call

```
    NPL.StartNetServer("127.0.0.1", "60001");
```

This must be done on both client and server. Please note that NPL does not distinguish between client and server. If you like, you may call the one who first calls `NPL.activate` to be client. For private client, the port number can be "0" to indicate that it will not be accessible by other computers, but it can always connect to other public NPL runtimes.

To make files within NPL runtime accessible by other NPL runtimes, one must add those files to public file lists by calling

```
    NPL.AddPublicFile("script/test/network/TestSimpleClient.lua", 1);
    NPL.AddPublicFile("script/test/network/TestSimpleServer.lua", 2);
```

the second parameter is id of the file. The file name to id map must be the same for all communicating computers. This presumption may be removed in futhure versions.

To create additional worker threads (NPL states) for handling messages, one can call

```
    for i=1, 10 do
        local worker = NPL.CreateRuntimeState("some_runtime_state"..i, 0);
        worker:Start();
    end
```

Each NPL runtime on the network (either LAN/WAN) is identified by a globally unique `nid` string. NPL itself does not verify or authenticate nid, it is the programmers' job to do it.

To add a trusted server address to nid map, one can call

```
    NPL.AddNPLRuntimeAddress({host="127.0.0.1", port="60001", nid="this_is_server_nid
↪"})
```

so that NPL runtime knows how to connection to the nid via TCP endpoints. host can be ip or domain name.

To activate a file on a remote NPL runtime(identified by a nid), one can call

```
while( NPL.activate("(some_runtime_state)this_is_server_nid:script/test/network/
↪TestSimpleServer.lua",
 {some_data_table}) ~=0 ) do
end
```

Please note that `NPL.activate()` function will return non-zero if no connection is established for the target, but it will immediately try to establish a new connection if the target address of server nid is known. The above sample code simply loop until activate succeed; in real world, one should use a timer with timeout or use one of the helper function such as `NPL.activate_with_timeout`.

When a file is activated, its activation function will be called, and data is inside the global msg table.

```
    local function activate()
        if(msg.nid) then
        elseif(msg.tid) then
        end
        NPL.activate(string.format("%s:script/test/network/TestSimpleClient.lua", msg.
↪nid or msg.tid), {some_date_sent_back})
    end
    NPL.this(activate)
```

`msg.nid` contains the nid of the source NPL runtime. `msg.nid` is nil, if connection is not authenticated or nid is not known. In such cases, `msg.tid` is always available; it contains the local temporary id. When the connection is authenticated, one can call following function to reassign nid to a connection or reject it. NPL runtime will try to reuse the same existing TCP connect between current computer and each nid or tid target, for all communcations between the two endpoints in either direction.

```
    NPL.accept(msg.tid, nid)  -- to rename tid to nid, or
    NPL.reject(msg.tid) -- to close the connection.
```

only the main thread support non-thread safe functions, such as those with rendering. For worker thread(runtime state), please only use NPL functions that are explicitly marked as thread-safe in the documentation.

NPL Extensibility and Scripting Performance

NPL provides three extensibility modes: (1) Lua scripting runtime states (2) Mono .Net runtimes (3) C++ plugin interface All of them can be used simultanously to create logics for game applications. All modes support cross-platform and multi-threaded computing.

Lua runtime is natively supported and has the most extensive ParaEngine API exposed. It is both extremely light weighted and having a good thread local memory manager. It is suitable for both client and server side scripting. It is recommended to use only lua scripts on the client side.

Mono .Net runtimes is supported via NPLMono.dll (which in turn is a C++ plugin dll). The main advantage of using .Net scripting is its rich libraries, popular language support(C#,Java,VB and their IDE) and compiled binary code performance. .Net scripting runtime is recommended to use on the server side only, since deploying .Net on the client requires quite some disk space. And .Net (strong typed language) is less effective than lua when coding for client logics.

C++ plugins allows us to treat dll file as a single script file. It has the best performance among others, but also is the most difficult to code. We would only want to use it for performance critical tasks or functions that make extensive use of other third-party C/C++ libraries. For example, NPLRouter.dll is a C++ plugin for routing messages on the server side.

Cross-Runtime Communication

A game project may have thousands of lua scripts on the client, mega bytes of C# code contained in several .NET dll assemblies on the server, and a few C++ plugin dlls on both client and server. (On linux, *.dll will actually find the *.so file) All these extensible codes are connected to the ParaEngine Runtime and have full access to the exposed ParaEngine API. Hence, source code written in different language runtimes can use the ParaEngine API to communicate with each other as well as the core game engine module.

More over, `NPL.activate` is the single most versatile and effective communication function used in all NPL extensibility modes. In NPL, a file is the smallest communication entity. Each file can receive messages either from the local or remote files.

- In NPL lua runtime, each script file with a activate() function assigned can receive messages from other scripts.
- In NPL .Net runtime, each C# file with a activate() function defined inside a class having the same name as the file name can receive messages from other scripts. (namespace around class name is also supported, see the example)
- In NPL C++ plugins, each dll file must expose a activate() function to receive messages from other scripts.

Following are example scripts and NPL.activate() functions to send messages to them.

```
-------------------------------------
File name:  script/HelloWorld.lua  (NPL script file)
activate:   NPL.activate("script/HelloWorld.lua", {data})
-------------------------------------
local function activate()
end
NPL.this(activate)

-------------------------------------
File name:  HelloWorld.cs inside C# mono/MyMonoLib.dll
activate:   NPL.activate("mono/MyMonoLib.dll/HelloWorld.cs", {data})
            NPL.activate("mono/MyMonoLib.dll/ParaMono.HelloWorld.cs", {data})
-------------------------------------
class HelloWorld
{
```

```
    public static void activate(ref int type, ref string msg)
    {
    }
}
namespace ParaMono
{
    class HelloWorld
    {
        public static void activate(ref int type, ref string msg)
        {
        }
    }
}
---------------------------------------
File name:  HelloWorld.cpp inside C++ MyPlugin.dll
activate:   NPL.activate("MyPlugin.dll", {data})
---------------------------------------
CORE_EXPORT_DECL void LibActivate(int nType, void* pVoid)
{
    if(nType == ParaEngine::PluginActType_STATE)
    {
    }
}
```

The `NPL.activate()` can use the file extension (*.lua, *.cs, *.dll) to distinguish the file type.

All `NPL.activate()` communications are asynchronous and each message must go through message queues between the sending and receiving endpoints. The performance of NPL.activate() is therefore not as efficient as local function calls, so they are generally only used to dispatch work tasks to different worker threads or communicating between client and server.Generally, 30000 msgs per second can be processed for 100 bytes message on 2.5G quad core CPU machine. For pure client side game logics, we rarely use NPL.activate(), instead we use local function calls whereever possible.

High Level Networking Programming Libraries

In real world applications, we usually need to build a network architecture on top of NPL, which usually contains the following tool sets and libraries.

- gateway servers and game(application) servers where the client connections are kept.
- routers for dispaching messagings between game(app) servers and database servers.
- database servers that performances business logics and query the underlying databases(My SQLs)
- memory cache servers that cache data for database servers to minimize access to database.
- Configuration files and management tools (shell scripts, server deployingment tools, monitoring tools, etc)
- backup servers for all above and DNS servers.

CHAPTER 19

Bootstrapping

Bootstrapping is to specify the first script file to load when starting NPL runtime, it will also be activated every 0.5 seconds. There are several ways to do it via command line.

The recommended way is to specify it explicitly with `bootstrapper=filename` parameters. Like below.

```
npl bootstrapper="script/gameinterface.lua"
```

If no file name is specified, `script/gameinterface.lua` will be used. A more concise way is to specify the file name as first parameter, like below.

```
npl helloworld.lua
```

Internally, the boot order of NPL is like below:

- When the NPL runtime starts, it first reads all of its command line arguments as name, value pairs and save them to an internal data structure for later use.
- NPL recognizes several predefined command names, one of them is bootstrapper. It specifies the main loop file.
- NPL initializes itself. This can take several seconds, involving reading configurations, graphics initialization, script API bindings, etc.
- When everything is initialized, the majority of NPL's Core API is available to the scripting interface and NPL loads the main loop script specified by the bootstrapper and calls its activation function every 0.5 seconds.
- Anything happens afterwards is up to the progammer who wrote the main loop script.

In addition to script file, one can also use XML file as the bootstrapper. The content of the xml should look like below.

```
<?xml version="1.0" ?>
<MainGameLoop>(gl)script/apps/Taurus/main_loop.lua</MainGameLoop>
```

To start it, we can use

```
npl script/apps/Taurus/bootstrapper.xml
```

Understanding File Path

When programming with NPL, the code will reference other script or asset file by relative file name. The search order of a file is as follows:

- check if file exists in the current development directory (in release time, this is the same as the working directory)

- check if file exists in the search paths (by default it is the current working directory)
- check if file exists in any loaded archive files (mainXXX.PKG or world/plugins ZIP files) in their loading order.
- if the file is NPL script file, look for precompiled binary file at ./bin/*filename*.o also in above places and order.
- if the file is listed in `asset_manifest.txt`, we will download it or open it from `./temp/cache` if already downloaded.
- check if disk file exists in all search paths of npl_packages.
- if none of above places are found, we will report file not found. Please also note, some API allows you to use specify search order, temporarily disable some places, or include certain directory like the current writable directory. Please refer to the usage of each API for details.

CHAPTER 20

Build-in NPL CommandLine

Native Params

The following command line is built-in to NPL executable.

- `bootstrapper="myapp/main.lua"`: set bootstrapper file
- [-d|D] [filename]: -d to run in background (daemon mode), -D to force run in fore ground.
- `dev="C:/MyDev/"`: set dev directory, if empty or "/", it means the current working directory.
- `servermode="true"`: disable 3D rendering and force running in server mode.
- `logfile="log2016.5.20.txt"`: change default `log.txt` location
- `single="true"`: force only one instance of the executable can be running in the OS.
- `loadpackage="folder1,folder1,..."`: commar separated list of packages to load before bootstrapping. Such as `loadpackage="npl_packages/paracraft/"`

NPL System Module Params

The following params are handled by the NPL system module

- `debug="main"` : enable IPC debugging to the main NPL thread. No performance penalties.
- `resolution="1020 680"`: force window size in case of client application

Paracraft Module Params

The following params are handled by the Paracraft Package

- `world="worlds/DesignHouse/myworld"` : Load a given world at start up.
- `mc="true"`: force paracraft (in case of running from MagicHaqi directory)

CHAPTER 21

NPL Load File

In NPL code, one may call something like `NPL.load("script/ide/commonlib.lua")` to load a file. All NPL code needs to be compiled in order to run. `NPL.load` does following things automatically for you.

- find the correct file either in NPL zip package or according to NPL search path defined in npl_packages and automatically use precompiled binary version (`*.o`) if available (see DeployGuide).
  - relative path is supported, but only recommended for private files, such as `NPL.load("./A.lua")` or `NPL.load("../../B.lua")`
  - when using relative path, file extension can be omitted, where *.npl and *.lua are searched, such as `NPL.load("./A")` or `NPL.load("../../B")`
- automatically resolve recursion.(such as File A load File B, while B also load A).
- compiles the script code if not yet compiled
  - if file extension is `*.npl`, NPL meta compiler will be invoked.
- The first time code is compiled, it also execute the code chunk in protected mode with `pcall`. In most cases, this means injecting new code to the global or exported table, so that one can use them later.
- it can also be used to load `C++/Mono C#/NPL packages`
- return the exported file module if any. See `file based module` section below for details.

```
/**
load a new file (in the current runtime state) without activating it. If the file is␣
↪already loaded,
it will not be loaded again unless bReload is true.
IMPORTANT: this function is synchronous; unlike the asynchronous activation function.
LoadFile is more like "include in C++".When the function returns, contents in the␣
↪file is loaded to memory.
@note: in NPL/lua, function is first class object, so loading a file means executing␣
↪the code chunk in protected mode with pcall,
in most cases,  this means injecting new code to the global table. Since there may be␣
↪recursions (such as A load B, while B also load A),
your loading code should not reply on the loading order to work. You need to follow␣
↪basic code injection principles.
For example, commonlib.gettable("") is the the commended way to inject new code to␣
↪the current thread's global table.
Be careful not to pollute the global table too much, use nested table/namespace.
Different NPL applications may have their own sandbox environments, which have their␣
↪own dedicated global tables, for example all `*.page` files use a separate global␣
↪table per URL request in NPL Web Server App.
@note: when loading an NPL file, we will first find if there is an up to date␣
↪compiled version in the bin directory. if there is,
```

```
we will load the compiled version, otherwise we will use the text version.  use bin␣
↪version, if source version does not exist; use bin version, if source and bin␣
↪versions are both on disk (instead of zip) and that bin version is newer than the␣
↪source version.
e.g. we can compile source to bin directory with file extension ".o", e.g. "script/
↪abc.lua" can be compiled to "bin/script/abc.o", The latter will be used if␣
↪available and up-to-date.
@param filePath: the local relative file path.
If the file extension is ".dll", it will be treated as a plug-in. Examples:
    "NPLRouter.dll"         -- load a C++ or C# dll. Please note that, in windows, it␣
↪looks for NPLRonter.dll; in linux, it looks for ./libNPLRouter.so
    "plugin/libNPLRouter.dll"           -- almost same as above, it is reformatted to␣
↪remove the heading 'lib' when loading. In windows, it looks for plugin/NPLRonter.
↪dll; in linux, it looks for ./plugin/libNPLRouter.so
@param bReload: if true, the file will be reloaded even if it is already loaded.
   otherwise, the file will only be loaded if it is not loaded yet.
@remark: one should be very careful when calling with bReload set to true, since this␣
↪may lead to recursive
    reloading of the same file. If this occurs, it will generate C Stack overflow␣
↪error message.
*/
NPL.load(filePath, bReload)
```

Code Injection Model

Since, in NPL/lua, table and functions are first class object, we use a very flexible code injection model to manage all dynamically loaded code during the life time of an application.

To inject new code, we use method like `commonlib.gettable`, `commonlib.inherit`, please see ObjectOriented for details. The developer should ensure they inject their code with unique names (such as `CompanyName.AppName.ModuleName` etc). In other words, do not pollute the global table.

To reference these code, the user needs to call `NPL.load` as well as `commonlib.gettable` to create a local stub variable in the file scope. Please note, due to `NPL.load` order, the stub may be created before the actual code is injected into it.

For example, you have a class file in `script/MyApp/MyClass.lua` like below

```
local MyClass = commonlib.inherit(nil, commonlib.gettable("MyApp.MyClass"));

MyClass.default_param = 1;

-- this is the constructor function.
function MyClass:ctor()
   self.map = {};
end

function MyClass:init(param1)
   self.map.param1 = param1;
   return self;
end

function MyClass:Clone()
   return MyClass:new():init(self:GetParam());
end
```

```
function MyClass:GetParam()
   return self.map.param1;
end
```

To use above class, one may use

```
NPL.load("(gl)script/MyApp/MyClass.lua")
local MyClass = commonlib.gettable("MyApp.MyClass");

local UserClass = commonlib.inherit(nil, commonlib.gettable("MyApp.UserClass"));

function UserClass:ctor()
   self.map = MyClass:new();  -- use another class
end
```

please see ObjectOriented for details.

File-based Modules

File-based modules use the containing source code filename to store exported object. So programmers do not need to explicitly specify object namespace in code, this solves several issues:

- code written in this style is independent of its containing file location.
- code looks more isolated.
- multiple versions of the same code can coexist.

  One not-so-convenient thing is that the user must explicitly load every referenced external modules written in this way in every file. This is the primary reason why the common NPL library are NOT written in this way.

Defining file-based module

There are three ways to define a file-based module. Suppose, you have a file called `A.npl` and you want to export some object from it.

- One way is to call `NPL.export` when file is loaded.

```
-- this is A.npl file
local A = {};
NPL.export(A);

function A.print(s)
  echo(s);
end
```

Following is a better and recommended way, which works with cyclic dependencies. See `Module Cyclic Reference` section for details.

```
-- this is A.npl file
local A = NPL.export();
function A.print(s)
  echo(s);
end
```

- Another way is simply return the object to be associated with the current file at the last line of code, like below. This is also a lua-compatible way. It does not work when there are cyclic dependencies.

```
-- this is A.npl file
local A = {};
return A;
```

- finally, there is an advanced manual way to simply add to the `_file_mod_` table.

```
-- this is A.npl file
local A = {};
_file_mod_[NPL.filename():gsub("([^/]*$)", "").."A.npl"] = A;
```

As you can see, file-based module simply automatically stores the mapping from the module's full filename to its exported object in a hidden global table. If one changes the filename or file location, the mapping key also changes. This is why you can load multiple versions of the same module and let the user to choose which one to use in a given source file.

Module Cyclic Reference

When you write file modules that depends on each file, there are cyclic dependencies. One workaround is to modify the default exported object, which is always an empty table, instead of creating a local table. See below. We have two files A and B that reference each other.

```
-- A.lua
local B = NPL.load("./B.lua")
local A = NPL.export();
function A.print(s)
  B.print(s)
end
```

```
-- B.lua
local A = NPL.load("./A.lua")
local B = NPL.export();
function B.print(s)
  echo(s..type(A));
end
```

> `NPL.export()` is the core of file module system in NPL, it will return the default empty table representing the current file. This is the same way as how `commonlib.gettable()` solve cyclic dependency.

Modules with Object Oriented Class Inheritance

The following shows an example of two class table with inheritance, and also cyclic dependency.

```
-- base_class.lua
local derived_class = NPL.load("./derived_class.lua")
local base_class = commonlib.inherit(nil, NPL.export());
function base_class:ctor()
   derived_class:cyclic_dependency_test();
end
```

```
-- derived_class.lua
local base_class = NPL.load("./base_class.lua")
local B = commonlib.inherit(base_class, NPL.export());
```

```
function B:ctor()
end
function B:cyclic_dependency_test()
end
```

Loading file-based module With module name

To import a file-based module, one calls `NPL.load(modname)`. `NPL.load` is a versatile function which can load either standard source files or file-based modules and return the exported module object.

> A `modname` is different from filename, a string is treated as `modname` if and only if it does NOT contain file extension or begin with "./" or "../". such as `NPL.load("sample_mod")`

`NPL.load(modname)` will automatically find the source file of the module by trying the following locations in order until it finds one. Using module name in NPL.load is less efficient than using explicit filename, because it needs to do the search every time it is called, so only use it at file-load time, such as at the beginning of a file.

- if the code that invokes `NPL.load` is from `npl_packages/[parentModDir]/`
  - try: `npl_packages/[parentModDir]/npl_mod/[modname]/[modname].lua|npl`
  - try: `npl_packages/[parentModDir]/npl_packages/[modname]/npl_mod/[modname]/[modname].l`
- try: `npl_mod/[modname]/[modname].npl|lua`
- try: `npl_packages/[modname]/npl_mod/[modname]/[modname].npl|lua`

Example 1: NPL.load("sample_mod") will test following file locations for both *.npl and *.lua file:

- if code that invokes it is from `npl_packages/[parentModDir]/`
  - `npl_packages/[parentModDir]/npl_mod/sample_mod/sample_mod.npl`
  - `npl_packages/[parentModDir]/npl_packages/sample_mod/npl_mod/sample_mod/sample_mod.n`
- `npl_mod/sample_mod/sample_mod.npl`
- `npl_packages/sample_mod/npl_mod/sample_mod/sample_mod.npl`

Example 2: NPL.load("sample_mod.xxx.yyy") will test following file locations for both *.npl and *.lua file:

- if code that invokes it is from `npl_packages/[parentModDir]/`
  - `npl_packages/[parentModDir]/npl_mod/sample_mod/xxx/yyy.npl`
  - `npl_packages/[parentModDir]/npl_packages/sample_mod/npl_mod/sample_mod/xxx/yyy.npl`
- `npl_mod/sample_mod/xxx/yyy.npl`
- `npl_packages/sample_mod/npl_mod/sample_mod/xxx/yyy.npl`

As you can see, the `npl_packages/` and `npl_mod/` are two special folders to search for when loading file based module. code inside npl_packages/xxx/ folder always use local `npl_mod` and local npl_packages before searching for global ones, this allow multiple versions of the same npl_mod to coexist in different npl_packages. It is up to the developer to decide how to package and release their application or modules.

Here is an example of npl_mod in main package

For more information on NPL packages, please see npl_packages

When to Use `require`

`require` is Lua's way to load a file based module. NPL hooks into this function to always call `NPL.load` with the same input before passing down. More specifically, NPL injects a custom loader (at index 2) into the lua's `package.loaders`. So calling `require` is almost identical to calling `NPL.load` so far, except that multiple code versions can not coexist.

Do not Use `require`

Since NPL.load now hooks original lua's require function. Facts in this section may not be true, but we still do not recommend using `require` in your own code, unless you are reusing third-party lua libraries. The reason is simple, NPL.load is backward-compatible with the original require, but the original require does not support features provided by NPL.load. Using require in your own code may confuse other lua developers. Therefore, code written for NPL developers should always use NPL.load.

One should only use `require` to load lua C plugins, do not use it in your own NPL scripts. `require` is rated low in community. Its original implementation is same, but use a similar global hidden table, whose logic you never know, in a flat manner. It also does not support cyclic dependency so far.

Moreover, some application may need to create sandbox environment to isolate code execution (code are injected to specified global table), using `require` give you no control of code injection.

Different NPL applications may have their own sandbox environments, which have their own dedicated global tables, for example all `*.page` files use a separate global table per URL request in NPL Web Server App.

Finally, require‘ is slow and uses different search paths than NPL; while NPL.load is fast and honors precompiled code either in package zip file or in search paths defined in npl_packages, and consistent on all platforms.

CHAPTER 22

File Activation

NPL can communicate with remote NPL script using the `NPL.activate` function. It is a powerful function, and is available in C++, mono plugin too.

Basic Syntax

Like in neural network, all communications are asynchronous and uni-directional without callbacks. Although the function does return a integer value. Please see `NPL Reference` for details.

```
NPL.activate(url, {any_pure_data_table_here, })
```

- @param `url`: a globally unique name of a NPL file name instance. The string format of an NPL file name is like below. `[(sRuntimeStateName|gl)][sNID:]sRelativePath[]` the following is a list of all valid file name combinations:
  - `user001@paraengine.com:script/hello.lua` – a file of user001 in its default gaming thread
  - `(world1)server001@paraengine.com:script/hello.lua` – a file of server001 in its thread world1
  - `(worker1)script/hello.lua` – a local file in the thread worker1
  - `(gl)script/hello.lua` – a glia (local) file in the current runtime state's thread
  - `script/hello.lua` – a file in the current thread. For a single threaded application, this is usually enough.
  - `(worker1)NPLRouter.dll` – activate a C++ file. Please note that, in windows, it looks for NPLRonter.dll; in linux, it looks for ./libNPLRouter.so
  - `(worker1)DBServer.dll/DBServer.DBServer.cs` – for C# file, the class must have a static activate function defined in the CS file.
- @param `msg`: it is a chunk of pure data table that will be transmitted to the destination file.

Neuron File

Only files associated with an activate function can be activated. This is done differently in NPL/C++/C# plugin

- In NPL, it is most flexible by using the build-in `NPL.this` function

```
local function activate()
   -- input is passed in a global "msg" variable
   echo(msg);
end
NPL.this(activate);
```

- `msg` is passed in a global `msg` variable which is visible to all files, and the `msg` variable will last until the thread receives the next activation message.
- In NPL's C++ plugin, you need to define a `C` function. Please see ParacraftSDK's example folder for details.
- In NPL's mono C# plugin, you simply define a class with a static `activate` function. Please see ParacraftSDK's example folder for details.

Input Message, `msg.nid` and `msg.tid`

In above code, `msg` contains the information received from the sender, plus the `source id` of the sender. For unauthenticated senders, the source id is stored in `msg.tid`, which is an auto-generated number string like "~1". The receiver can keep using this temporary id `msg.tid` to send message back, such as

```
local function activate()
   -- input is passed in a global "msg" variable
   NPL.activate(format("%s:some_reply_file.lua", msg.tid or msg.nid), {"replied"});
end
NPL.this(activate);
```

The receiver can also rename the temporary `msg.tid` by calling `NPL.accept(msg.tid,nid_name)`, so the next time if the receiver got a message from the same sender (i.e. the same TCP connection), the `msg.nid` contains the last assigned name and `msg.tid` field no longer exists. We usually use `NPL.accept` to distinguish between authenticated and unauthenticated senders, and reject unauthenticated messages by calling `NPL.reject(msg.tid)` as early as possible to save CPU cycles.

See following example:

```
local function activate()
   -- input is passed in a global "msg" variable
   if(msg.tid) then
      -- unauthenticated? reject as early as possible or accept it.
      if(msg.password=="123") then
         NPL.accept(msg.tid, msg.username or "default_user");
      else
         NPL.reject(msg.tid);
      end
   elseif(msg.nid) then
      -- only respond to authenticated messages.
      NPL.activate(format("%s:some_reply_file.lua", msg.nid), {"replied"});
   end
end
NPL.this(activate);
```

Please note, `msg.tid or msg.nid` always match to a single low-level TCP connection, hence their names are shared to all neuron files in the process. For example, if you accept in one neuron file, all other neuron files will receive messages in `msg.nid` form.

Please see

Neuron File Visibility

For security reasons, all neuron files can be activated by other files in the same process. This includes scripts in other local threads of the same process. See also MultiThreading.

To expose script to remote computers, one needs to do two things.

- one is to start NPL server by listening to an IP and port: NPL uses TCP protocol for all communications.
- second is to tell NPL runtime that a given file is a `public neuron file`.

See below:

```
NPL.StartNetServer("0.0.0.0", 8080);
NPL.AddPublicFile(filename, id);
```

- "0.0.0.0" means all IP addresses, one can also use "127.0.0.1", "localhost" or whatever IP addresses.
- 8080: is the port number. Pick any one you like.

The second parameter to `NPL.AddPublicFile` is an integer, which is transmitted on behalf of the long filename to save bandwidth. So it must be unique if you add multiple public files.

> Please note, that file name must be relative to working directory. such as `NPL.AddPublicFile("script/test/test.lua",1)`. Absolute path is not supported at the moment.

Activating remote file

Once a server NPL runtime exposes a public file, other client NPL runtime can activate it with the `NPL.activate` function. Please note that, an NPL runtime can be both server and client. The one who started the connection is usually called server. Pure client must also call `NPL.StartNetServer` in order to activate the server. But it can specify `port="0"` to indicate that it will not listen for incoming connections.

However, on the client, we need to assign a local name to the remote server using the `NPL.AddNPLRuntimeAddress`, so that we can refer to this server by name in all subsequent `NPL.activate` call.

```
NPL.AddNPLRuntimeAddress({host = "127.0.0.1", port = "8099", nid = "server1"})
```

Usually we do this during initialization time only once. After that we can activate public files on the remote server like below:

```
NPL.activate("server1:helloworld.lua", {})
```

Please note the name specified by `nid` is arbitrary and used only on the client computer to refer to a computer. In other words, different clients can name the same remote computer differently.

Message Delivery Guarantees

Please note that the first time that a computer activate a remote file, a TCP connection is automatically established, but the first message is NOT delivered. This is because `NPL.activate()` is asynchronous, it must return a value before the new connection is established. It always returns 0 when your message is delivered via an existing path, and non-zero in case of first message to a remote system.

If there is no already established path (i.e no TCP connection), NPL will immediately try to establish it. However, please note, the message that returns non-zero is NOT delivered, even if NPL successfully established a path to the remote system soon afterwards. Thus it is the programmer's job to activate again until `NPL.activate` returns 0. This guarantees that a message with 0 return value is always delivered at least in the viewpoint of NPL runtime.

The same mechanism can be used to recover lost-connections.

To write fault-tolerant message passing code, consider following.

- When a NPL runtime process start, ping remote process with NPL.activate until it returns 0 to establish TCP connections. This ensures that all subsequent NPL.activate across these two systems can be delivered.
- Discover Lost Connections:
  - Method1: Use a timer to ping or listen for disconnect system event and reconnect in case connection is lost. However, individual messages may be lost during these time.
  - Method2: Use a wrapper function to call NPL.activate, which checks its return value. If it is non-zero, either reconnect with timeout or put message to a pending queue in case connection can be recovered shortly and resend queued messages.

We leave it to the programmer to handle all occasions when NPL.activate returns non-zero values, since different business logic may use a different approach.

Example Client/Server app

To run the example, call following.

```
npl "script/test/network/SimpleClientServer.lua" server="true"
npl "script/test/network/SimpleClientServer.lua" client="true"
```

The source code of this demo is also included in `ParaCraftSDK/examples` folder.

filename: `script/test/network/SimpleClientServer.lua`

```
--[[
Author: Li,Xizhi
Date: 2009-6-29
Desc: start one server, and at least one client.
-----------------------------------------------
npl "script/test/network/SimpleClientServer.lua" server="true"
npl "script/test/network/SimpleClientServer.lua" client="true"
-----------------------------------------------
]]
NPL.load("(gl)script/ide/commonlib.lua"); -- many sub dependency included

local nServerThreadCount = 2;
local initialized;
local isServerInstance = ParaEngine.GetAppCommandLineByParam("server","false") ==
↪"true";

-- expose these files. client/server usually share the same public files
local function AddPublicFiles()
    NPL.AddPublicFile("script/test/network/SimpleClientServer.lua", 1);
end

-- NPL simple server
local function InitServer()
    AddPublicFiles();
```

```
    NPL.StartNetServer("127.0.0.1", "60001");

    for i=1, nServerThreadCount do
        local rts_name = "worker"..i;
        local worker = NPL.CreateRuntimeState(rts_name, 0);
        worker:Start();
    end

    LOG.std(nil, "info", "Server", "server is started with %d threads",␣
↪nServerThreadCount);
end

-- NPL simple client
local function InitClient()
    AddPublicFiles();

    -- since this is a pure client, no need to listen to any port.
    NPL.StartNetServer("0", "0");

    -- add the server address
    NPL.AddNPLRuntimeAddress({host="127.0.0.1", port="60001", nid="simpleserver"})

    LOG.std(nil, "info", "Client", "started");

    -- activate a remote neuron file on each thread on the server
    for i=1, nServerThreadCount do
        local rts_name = "worker"..i;
        while( NPL.activate(string.format("(%s)simpleserver:script/test/network/
↪SimpleClientServer.lua", rts_name),
            {TestCase = "TP", data="from client"}) ~=0 ) do
            -- if can not send message, try again.
            echo("failed to send message");
            ParaEngine.Sleep(1);
        end
    end
end

local function activate()
    if(not initialized) then
        initialized = true;
        if(isServerInstance) then
            InitServer();
        else
            InitClient();
        end
    elseif(msg and msg.TestCase) then
        LOG.std(nil, "info", "test", "%s got a message", isServerInstance and "server
↪" or "client");
        echo(msg);
    end
end
NPL.this(activate);
```

The above server is actually multi-threaded, please see MultiThreading for details.

> The first time, `NPL.activate` calls a new remote server (with which we have not established TCP connection), the message is dropped and returned a non-zero value. `NPLExtension.lua`

contains a number of helper functions to help you for sending a guaranteed message, such as `NPL.activate_with_timeout`.You need to include commonlib to use it.

Trusted Connections and NID

In the receiver's activate function, it can assign any name or `nid` to incoming connection's NPL runtime.

HelloWorld Example

Now, here comes a more complicated helloworld. It turns an ordinary `helloworld.lua` file into a neuron file, by associating an `activate` function with it. The file is then callable from any NPL thread or remote computer by its NPL address(url).

```
local function activate()
   if(msg) then
      print(msg.data or "");
   end
   NPL.activate("(gl)helloworld.lua", {data="hello world!"})
end
NPL.this(activate);
```

Simple Web Server Example

NPL uses a HTTP-compatible protocol, so it is possible to handle standard HTTP request using the same NPL server. When NPL runtime receives a HTTP request message, it will send it to a publicly visible file with id `-10`. So we can create a simple HTTP web server with just a number of lines, like below:

filename: `main.lua`

```
NPL.load("(gl)script/ide/commonlib.lua");

local function StartWebServer()
    local host = "127.0.0.1";
    local port = "8099";
    -- tell NPL runtime to route all HTTP message to the public neuron file `http_
 ↪server.lua`
    NPL.AddPublicFile("source/SimpleWebServer/http_server.lua", -10);
    NPL.StartNetServer(host, port);
    LOG.std(nil, "system", "WebServer", "NPL Server started on ip:port %s %s", host,␣
 ↪port);
end
StartWebServer();

local function activate()
end
NPL.this(activate);
```

filename: `http_server.lua`

```
NPL.load("(gl)script/ide/Json.lua");
NPL.load("(gl)script/ide/LuaXML.lua");
```

```
local tostring = tostring;
local type = type;

local npl_http = commonlib.gettable("MyCompany.Samples.npl_http");

-- whether to dump all incoming stream;
npl_http.dump_stream = false;

-- keep statistics
local stats = {
    request_received = 0,
}

local default_msg = "HTTP/1.1 200 OK\r\nContent-Length: 31\r\nContent-Type: text/
↪html\r\n\r\n<html><body>hello</body></html>";

local status_strings = {
    ok ="HTTP/1.1 200 OK\r\n",
    created ="HTTP/1.1 201 Created\r\n",
    accepted ="HTTP/1.1 202 Accepted\r\n",
    no_content = "HTTP/1.1 204 No Content\r\n",
    multiple_choices = "HTTP/1.1 300 Multiple Choices\r\n",
    moved_permanently = "HTTP/1.1 301 Moved Permanently\r\n",
    moved_temporarily = "HTTP/1.1 302 Moved Temporarily\r\n",
    not_modified = "HTTP/1.1 304 Not Modified\r\n",
    bad_request = "HTTP/1.1 400 Bad Request\r\n",
    unauthorized = "HTTP/1.1 401 Unauthorized\r\n",
    forbidden = "HTTP/1.1 403 Forbidden\r\n",
    not_found = "HTTP/1.1 404 Not Found\r\n",
    internal_server_error = "HTTP/1.1 500 Internal Server Error\r\n",
    not_implemented = "HTTP/1.1 501 Not Implemented\r\n",
    bad_gateway = "HTTP/1.1 502 Bad Gateway\r\n",
    service_unavailable = "HTTP/1.1 503 Service Unavailable\r\n",
};
npl_http.status_strings = status_strings;

-- make an HTML response
-- @param return_code: nil if default to "ok"(200)
function npl_http.make_html_response(nid, html, return_code, headers)
    if(type(html) == "table") then
        html = commonlib.Lua2XmlString(html);
    end
    npl_http.make_response(nid, html, return_code, headers);
end

-- make a json response
-- @param return_code: nil if default to "ok"(200)
function npl_http.make_json_response(nid, json, return_code, headers)
    if(type(html) == "table") then
        json = commonlib.Json.Encode(json)
    end
    npl_http.make_response(nid, json, return_code, headers);
end

-- make a string response
-- @param return_code: nil if default to "ok"(200)
-- @param body: must be string
-- @return true if send.
```

```
function npl_http.make_response(nid, body, return_code, headers)
    if(type(body) == "string" and nid) then
        local out = {};
        out[#out+1] = status_strings[return_code or "ok"] or return_code["not_found"];
        if(body~="") then
            out[#out+1] = format("Content-Length: %d\r\n", #body);
        end
        if(headers) then
            local name, value;
            for name, value in pairs(headers) do
                if(name ~= "Content-Length") then
                    out[#out+1] = format("%s: %s\r\n", name, value);
                end
            end
        end
        out[#out+1] = "\r\n";
        out[#out+1] = body;

        -- if file name is "http",  the message body is raw http stream
        return NPL.activate(format("%s:http", nid), table.concat(out));
    end
end


local function activate()
    stats.request_received = stats.request_received + 1;
    local msg=msg;
    local nid = msg.tid or msg.nid;
    if(npl_http.dump_stream) then
        log("HTTP:"); echo(msg);
    end
    npl_http.make_response(nid, format("<html><body>hello world. req: %d. input is %s
↪</body></html>", stats.request_received, commonlib.serialize_compact(msg)));
end
NPL.this(activate)
```

For a full-fledged build-in HTTP server framework in NPL, please see WebServer

CHAPTER 23

Networking

This topic is FAQ to networking API, please see Activation File for real guide of remote communications.

In NPL, you do not need to create any sockets or writing message loops for communication with remote computers. Instead, all communications are managed by NPL very efficiently (see NPLArchitecture for details). Users only need one single function to establish connection, authenticate and communicate with remote computers.

Initialize Networking Interface

By default, NPL is started without listening on any port, unless you called some library, such as the WebServer. To enable networking in NPL, one needs to call

```
-- a server that listen on 8080 for all IP addresses
NPL.StartNetServer("0.0.0.0", 8080);
```

The client also needs to call `NPL.StartNetServer`.

```
-- a client that does not listen on any port. Just start the network interface.
NPL.StartNetServer("127.0.0.1", "0");
```

Please see Activation File for details.

Server Parameters

The following is from WebServer‘s config file. You can have a general picture of what is configurable in NPL.

```
    <!--HTTP server related-->
    <table name='NPLRuntime'>
      <!--whether to use compression for incoming connections. This must be true in␣
↪order for CompressionLevel and CompressionThreshold to take effect-->
      <bool name='compress_incoming'>true</bool>
      <!---1, 0-9: Set the zlib compression level to use in case compresssion is␣
↪enabled.
      Compression level is an integer in the range of -1 to 9.
          Lower compression levels result in faster execution, but less compression.␣
↪Higher levels result in greater compression,
          but slower execution. The zlib constant -1, provides a good compromise␣
↪between compression and speed and is equivalent to level 6.-->
      <number name='CompressionLevel'>-1</number>
      <!--when the NPL message size is bigger than this number of bytes, we will use␣
↪m_nCompressionLevel for compression.
```

```
        For message smaller than the threshold, we will not compress even m_
↪nCompressionLevel is not 0.-->
     <number name='CompressionThreshold'>204800</number>
     <!--if plain text http content is requested, we will compress it with gzip when␣
↪its size is over this number of bytes.-->
     <number name='HTTPCompressionThreshold'>12000</number>
     <!--the default npl queue size for each npl thread. defaults to 500. may set to␣
↪something like 5000 for busy servers-->
     <number name='npl_queue_size'>20000</number>
     <!--whether socket's SO_KEEPALIVE is enabled.-->
     <bool name='TCPKeepAlive'>true</bool>
     <!--enable application level keep alive. we will use a global idle timer to␣
↪detect if a connection has been inactive for IdleTimeoutPeriod-->
     <bool name='KeepAlive'>false</bool>
     <!--Enable idle timeout. This is the application level timeout setting.-->
     <bool name='IdleTimeout'>false</bool>
     <!--how many milliseconds of inactivity to assume this connection should be␣
↪timed out. if 0 it is never timed out.-->
     <number name='IdleTimeoutPeriod'>1200000</number>
     <!--queue size of pending socket acceptor-->
     <number name='MaxPendingConnections'>1000</number>
   </table>
```

Please note, the `IdleTimeout` should be set for both client and server, because timeout may occur on either side, see below for best timeout practice. `TCPKeepAlive` is only for server side.

To programmatically set parameters, see following example:

```
local att = NPL.GetAttributeObject();
att:SetField("TCPKeepAlive", true);
att:SetField("KeepAlive", false);
att:SetField("IdleTimeout", false);
att:SetField("IdleTimeoutPeriod", 1200000);
NPL.SetUseCompression(true, true);
att:SetField("CompressionLevel", -1);
att:SetField("CompressionThreshold", 1024*16);
-- npl message queue size is set to really large
__rts__:SetMsgQueueSize(5000);
```

TCP Keep Alive

If you want a persistent already-authenticated connection even there are no messages sent for duration longer than 20 seconds, you probably want to generate some kind of regular ping messages. Because otherwise, either the server or the client may consider the TCP connection is dead, and you will lose your authenticated session on the server side.

Although NPL allows you to configure server side application level ping messages globally with `KeepAlive` attribute, it is not recommended to enable it on the server or set its value to a very large value (such as several minutes).

The recommended way is that the client should initiate the ping messages to all remote processes on regular intervals.

For a robust client application, the client also needs to handle connection lost and recover or re-authenticate at application level.

CHAPTER 24

Multi-Threading

In NPL, multi-threading is handled in the same way as networking communication. In other words, activating scripts in other local threads is virtually the same as calling scripts running on another computer. You simply communicate with remote script file with the `NPL.activate` in the same way for both local threads and remote computers. The only difference is that the target script url is different.

For example, to activate a script on local thread `A`, one can use

```
NPL.activate("(A)helloworld.lua", {});
```

To activate the script in a remote computer `B`'s `C` thread, one can use

```
NPL.activate("(C)B:helloworld.lua", {});
```

For more information, please see file activation

Creating NPL Worker Thread

NPL worker thread must be created, before messages to it can be processed by script running in that thread.

`NPL.activate("(A)helloworld.lua",{});` does not automatically create thread `A`. You need to call following code to create NPL runtime on thread `A`.

```
NPL.CreateRuntimeState("A", 0):Start();
```

After that, messages sent to `(A)helloworld.lua` will be processed in a real system-level thread called `A`.

Example Project

Now let us create a real multi-threaded application with just a single file `script/test/TestMultithread.lua`. The application print `helloworld` in 5 threads simultaneously.

```
NPL.load("(gl)script/ide/commonlib.lua");

local function Start()
   for i=1, 5 do
      local thead_name = "T"..i;
      NPL.CreateRuntimeState(thead_name, 0):Start();
      NPL.activate(format("(%s)script/test/TestMultithread.lua", thead_name), {
```

```
       text = "hello world",
       sleep_time = math.random()*5,
      });
   end
end

local isStarted;
local function activate()
   if(msg and msg.text) then
      -- sleep random seconds to simulate heavy task
      ParaEngine.Sleep(msg.sleep_time);
      LOG.std(nil, "info", "MultiThread", "%s from thread %s", msg.text,  __rts__:
↪GetName());
   elseif(not isStarted) then
      -- initialize on first call
      isStarted = true;
      Start();
   end
end

NPL.this(activate);
```

To run above file, use

```
NPL.activate("(gl)script/test/TestMultithread.lua");
```

or from command line

```
npl script/test/TestMultithread.lua
```

The output will be something like below

```
2016-03-16 6:22:00 PM|T1|info|MultiThread|hello world from thread T1
2016-03-16 6:22:01 PM|T3|info|MultiThread|hello world from thread T3
2016-03-16 6:22:03 PM|T2|info|MultiThread|hello world from thread T2
2016-03-16 6:22:03 PM|T5|info|MultiThread|hello world from thread T5
2016-03-16 6:22:04 PM|T4|info|MultiThread|hello world from thread T4
```

Advanced Examples

Following NPL modules utilize multiple local threads.

- `script/apps/DBServer/DBServer.lua`: a database server, each thread for processing SQL logics, a thread monitor is used to find the most free thread to route any sql query.

Other Options

By design, threading should be avoided to simplify software design and debugging. In addition to real threads, it is usually preferred to use architecture to avoid using thread at all.

- `Timer/callbacks/events/signals` are good candidates for asynchronous tasks in a single thread. With `NPL.activate`, it even allows you to switch implementation without changing any code; and you can boost performance or debug code in a single thread more easily.

- `Coroutine` is a lua language feature, which is also supported by NPL. In short, it uses a single thread to simulate multiple virtual threads, allowing you to share all data in the same thread without using locks, but still allowing the developer to `yield` CPU resource to other virtual threads at any time. The interactive debugging module in NPL is implemented with coroutines. Please see `script/ide/Debugger/IPCDebugger.lua` for details.

CHAPTER 25

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 26

NPL Common Libraries

NPL common Libraries contains a rich and open source collection of libraries covering: io, networking, 2d/3d graphics, web server, data structure, and many other software frameworks.

Library Installation

NPL common libraries can be installed in `pkg` or `zip` or in plain source code. When you deploy your application, you can choose to use our pre-made zip file or deploy only used files in a zip package by yourself.

If you installed NPL runtime via ParaCraftSDK on windows platform, you will already have NPL common library installed in `ParaCraftSDKGit/NPLRuntime/win/packages/main.pkg`.

If you installed NPL runtime from source code on linux. It does not have NPL common libraries preinstalled. Instead you need to copy or link the `script` folder to your project's working directory. The folder to link to can be found here, which contains thousands of open source and documented files all written in NPL.

Major Libraries

For minimum server-side application development, include this:

```
NPL.load("(gl)script/ide/commonlib.lua");
```

or

```
NPL.load("(gl)script/ide/System/System.lua");
```

For full-fledged heavy 2d/3d application development, include this:

```
NPL.load("(gl)script/ide/IDE.lua");
```

or

```
NPL.load("(gl)script/kids/ParaWorldCore.lua");
```

Documentation

NPL libraries are themselves written in pure NPL scripts. They usually have no external dependencies, but on the low level NPL API provided by NPL runtime. These API is in turn implemented in C/C++ in cross-platform ways.

> The documentation for low level API is here https://codedocs.xyz/LiXizhi/NPLRuntime/modules.html

However, it is not particularly useful to read low level API. All you need to do it is to glance over it, and then dive into the source code of NPL libraries here.

CHAPTER 27

Object Oriented Programming

All NPL library source code is written in object oriented way. It is recommended that you do the same for your own code.

> Lua itself can be used as a functional language, which does not enforce object oriented programming. It is the programmer's choice of how to write their code.

NPL has a number of library files to help you code in object oriented ways.

Referencing a class

`commonlib.gettable` and `NPL.load` is the primary way for you to include and import other component. Because all NPL libraries are defined to be compatible with `commonlib.gettable`, it is possible to import the class namespace table without loading it. This also makes the order of importing or loading libraries trivial.

Remember that NPL/Lua is a statically scoped language and each function is hash value on its containing table, it is important to cache class table on a local variable.

Example of importing `commonlib.LinkedList` into `LinkedList`.

```
local LinkedList = commonlib.gettable("commonlib.LinkedList");

local MyClass = commonlib.gettable("MyApp.MyClass");
function MyClass.Test()
   local list = LinkedList:new();
end
```

By the time, you call the function on the imported table, you need to `NPL.load` its implementation before hand. As you see, most common class is already buddled in common include files, such as `NPL.load("(gl)script/ide/commonlib.lua");`, this include the implementation of `commonlib.gettable` as well. If you are interested, you can read its source code. So if you have loaded that file before, such as in your `bootstrapper` file, you do not need to load it in every other files. But you need to call `commonlib.gettable` on the beginning of every file using the given library, for better performance.

> It is good practice to only `NPL.load` non-frequently used files shortly before they are used. So that, your application can boot up faster, and use less memory.

Defining a singleton class

For singleton class, which contains only static functions. It is sufficient to use `commonlib.gettable` to define the class. The idea is that you reference the class,and then add some static implementations to it dynamically.

```
local MyClass = commonlib.gettable("MyApp.MyClass");

function MyClass.Method1()

end

function MyClass.Method2()

end
```

Defining an ordinary class

For class, which you want to create instances from, you need to use the `commonlib.inherit` function. For implementation of `commonlib.inherit`, please see script/ide/oo.lua.

The following will define a `MyClass` class with `new` method.

```
local MyClass = commonlib.inherit(nil, commonlib.gettable("MyApp.MyClass"));

MyClass.default_param = 1;

-- this is the constructor function.
function MyClass:ctor()
   self.map = {};
end

function MyClass:init(param1)
   self.map.param1 = param1;
   return self;
end

function MyClass:Clone()
   return MyClass:new():init(self:GetParam());
end

function MyClass:GetParam()
   return self.map.param1;
end
```

To create a new instance of it

```
local MyClass = commonlib.gettable("MyApp.MyClass");

local c1 = MyClass:new():init("param1");
local c2 = MyClass:new():init("param1");
```

You can define a derived class like below

```
local MyClassDerived = commonlib.inherit(commonlib.gettable("MyApp.MyClass"),␣
↪commonlib.gettable("MyApp.MyClassDerived"));

-- this is the constructor function.
function MyClassDerived:ctor()
   -- parent class's ctor() have been automatically called
end

function MyClassDerived:init(param1)
   MyClassDerived._super.init(self, param1);
   return self;
end
```

Advanced `ToolBase` class

If you want a powerful class with `event/signal/auto property`, and `dynamic reflections`, you can derive your class from `ToolBase` class.

See the examples.

```
local Rect = commonlib.gettable("mathlib.Rect");

-- class new class
local UIElement = commonlib.inherit(commonlib.gettable("System.Core.ToolBase"),␣
↪commonlib.gettable("System.Windows.UIElement"));
UIElement:Property("Name", "UIElement");
UIElement:Signal("SizeChanged");
UIElement:Property({"enabled", true, "isEnabled", auto=true});


function UIElement:ctor()
    -- client rect
    self.crect = Rect:new():init(0,0,0,0);
end

function UIElement:init(parent)
    self:SetParent(parent);
    return self;
end

-- many other functions omitted here
```

CHAPTER 28

Deploy Your Application

You can deploy your application to windows, linux, ios, android, etc. Both 32bits/64bits versions are supported.

- NPL scripts can be deployed in plain `*.lua or *.npl` text files or in precompiled `*.o` binary file.
- You can also buddle all or part of your scripts or any other read-only resource files into one or more zipped archive files.
- You can deploy NPL runtime globally or side-by-side with your application files.

For automatic deployment, please install and use `ParacraftSDK`. This articles explains how to do it manually.

Pre-compiling NPL script

Precompiled NPL script is called bytecode in lua. The bytecode generated by `Luajit` is incompatible with bytecode generated by `lua`, but is cross-platform for any (32bits/64bits) architecture. If you choose to deploy your app with bytecode, you must also choose to use luajit or lua when deploying NPL runtime.

Please read the documentation in `NPLCompiler.lua` for more information. Examples:

```
NPL.load("(gl)script/ide/Debugger/NPLCompiler.lua");
NPL.CompileFiles("script/*.lua", nil, 100);
```

Buddle Scripts in Zip files

You can buddle your script and assets in zip files. There are two kinds of zip files: one is the standard `*.zip` file, the other is called `*.pkg` file. You can create `*.pkg` file from a standard zip file with the NPL runtime like below. `*.pkg` use a simple encription algorithm over the zip file.

```
ParaAsset.GeneratePkgFile("main.zip", "main.pkg");
```

When application starts, NPL runtime will automatically load all `main*.pkg` and `main*.zip` files in the application's start directory into memory. The load order is based on file name, so that the a file in "main_patch2.pkg" will overwrite the same file in "main_patch1.pkg".

Please note that loading `pkg` file is very fast, it only copys the file into memory, individual script file or assets are only unzipped and parsed on first use.

A programmer can also programmatically load or unload any archive file using the NPL API like below.

```
NPL.load("pluginABC.pkg");
-- or using explicit calls
ParaAsset.OpenArchive("pluginABC.pkg", true);
```

The second parameter is whether to use relative path in archive files. (i.e. file path in archive file are relative to the containing directory). Search paths, such as from npl_packages are honored when loading archives.

Deploy NPLRuntime Side-By-Side

Deploying NPL Runtime side-by-side is as easy as copying all executable files to the application directory. The recommended deployment folder structures is below

```
bin/: npl exe, dll, lua,luajit, etc
packages/: common *.pkg *.zip package files
script/: your own script files
config.txt
any other files
```

Another way is to deploy everything to the application root directory.

```
script/: your own script files
npl exe, dll, lua,luajit, etc
common *.pkg *.zip package files
config.txt
any other files
```

> if `config.txt` file is on the root application directory. Its cmdline content will be appended to NPL command line when running NPL runtime from this directory.

An example `config.txt`, see below:

```
cmdline=noupdate="true" debug="main" bootstrapper="script/apps/HelloWorld/main.lua"
```

Luajit vs Lua

Luajit and Lua are ABI-compatible, meaning that you can deploy NPL runtime with them simply by replacing `lua.dll(so)` with either implementation. Luajit is the recommended way for release deployment on all platforms. However, currently on iOS, Luajit is used but disabled, since it is not allowed. Even a disabled luajit runs faster than lua and its bytecode is the same for both 32bits and 64bits OS. So you would only want to deploy lua dll for debugging purposes on development machines.

CHAPTER 29

NPL Packages

NPL Package is a special folder under `npl_packages/[package_name]/`. Files in it are always organized as if they are relative to the working directory. So this folder can be used as a search path directly, or zipped to a `*.zip|pkg` to be used as an archive file, or loaded by file module name all at the same time.

NPL package serves following purposes:

- it provides a way to buddle and release multiple software modules to be used by someone else.
- it can be used as additional search path for third-party plugins, which I will explain later.
- it provides a way to allow different versions of the same file module to coexist by putting them in different npl packages in a single application. See LoadFile
- it provides a way to install third-party modules at development time.
- it provides a way to share big asset files (like textures, 3d models, audios) between modules, because all npl_packages share the same working directory.

Package As Search Path

Files in `npl_packages/[package_name]` are usually relative to the root working directory.

Therefore, developer who wants to use other people's modules can simply add `npl_packages/[package_name]` to global search path by calling following code:

```
NPL.load("npl_packages/some_test_module/")
```

The trick is to end with `/` in file name. What this function does is actually find the package folder in a number of possible locations (see next section), and then add it to the global search path. By default, when NPL starts, it will always try to load the official 'npl_packages/main/' package. So you do not need to call `NPL.load("npl_packages/main/")` in order to use the rich set of open source NPL libraries.

Another way is to load via command line, such as below. See NPLCommandLine

```
npl loadpackage="npl_packages/paracraft/" dev="/"
```

How NPL Locate Packages

NPL locates package folder given by its relative path in following order:

- search in current working directory

- search in current executable directory
- search recursively for 5 parent directories of the executable directory.

For example, suppose:

- your current working directory is `/home/myapp/`,
- and your executable directory is `/opt/NPLRuntime/redist/bin64/`,

then `NPL.load("npl_packages/main/")` will search following directories until one exists and add it to the search path.

- /home/myapp/npl_packages/main/
- /opt/NPLRuntime/redist/bin64/npl_packages/main/
- /opt/NPLRuntime/redist/npl_packages/main/
- /opt/NPLRuntime/npl_packages/main/
- /opt/npl_packages/main/
- /npl_packages/main/

What Happened After Loading A Package

The short answer is `nothing happens`, because loading a package only add its folder to the global search path. You still need to load any module files in the package folder with `NPL.load`. For example, support you have `NPL.load("npl_packages/test/")` successfully loaded at '/home/myapp/npl_packages/test/' and there is module file at `/home/myapp/npl_packages/test/script/any_module_folder/test.lua` Then you can load it with relative file path. `NPL.load("script/any_module_folder/test.lua")` However, if there is a file at your working directory such as `/home/myapp/script/any_module_folder/test.lua`, this file will be loaded rather than the one in the global search path.

File Search Order

Files in current working directory are always searched first (including those in zipped archive files) before we resolve to global search path. Moreover, search path added last is actually searched first. There is one exception, if NPL is run with dev directory in its command line `dev="dev_folder"`, NPL packages in `dev_folder` are loaded before zipped archive files. This allows one to use the latest source code during development.

For example:

```
NPL.load("npl_packages/A/")
NPL.load("npl_packages/B/")
NPL.load("test.lua");
```

`test.lua` is searched first in current working directory and any loaded archive (zip, pkg) file. If not exist, it will search in `npl_packages/B/` and then in `npl_packages/A/`.

Usage

For package developers:

NPL packages are mostly used at development time by other developers (not end users).

i.e. If you want other developers' to use your code, you can upload your working directory to git, so that other developers can clone your project to their `npl_packages/[your module name]`. Package can include not only source code, but also binary asset files, like images, textures, sound, 3d models, etc.

For other developers:

> Other developers should merge all used npl_packages to working directory, before releasing their software.

i.e. It is the developer's job to resolve dependencies, in case multiple versions of the same file exist among used npl_packages. At release time, it is recommended NOT to redistribute the npl_package folder, but copy/merge the content in them to the working directory, pre-compile all source code and package code and/or assets in one or multiple archive files. Please see the DeployGuide for details. However, if different versions of the same file must coexist, we can use file-based modules to distribute in separate `npl_packages` folders.

Where To Find NPL Packages

Each repository under NPLPackages is a valid npl_package managed by the community.

> Click here for more details on NPL packages.

If one want to upload their own package here, please make an issue here, and provide links to its code.

How To Install A Package

Simply create a folder under your development's working directory, create a sub folder called `npl_packages`. And then run git clone from there. Like this:

```
cd npl_packages
git clone https://github.com/NPLPackages/main.git
```

See also paracraft package for another example.

File-based Modules

See LoadFile.

How to Contribute

It is NOT advised to modify or add files in the ./npl_packages folder, instead create a similar directory structure in your project's development directory if you want to add or modify package source code. If you do want to contribute to any npl packages, please fork it on github and send pull requests to its author on github.

For example, if you want to modify or add a file like `./npl_packages/main/.../ABC.lua` Instead of modify it in the npl package folder, you simply create a file at the root development folder with the same directory structure like this `./.../ABC.lua`. At runtime, your version of file will be loaded instead of the one in npl package folder.

When your code is mature, you may consider fork the given npl_package in another place, and merge your changed files and send the author a pull request. If the author responds fast, he or she may accept your changes and you can later get rid of your changed files in your original project.

CHAPTER 30

Meta Programming in NPL

Meta programming allows you to extend NPL syntax using the NPL language itself. For example, following code are valid in NPL.

```
local a=1;
loop(){  please execute the code 10 times with i
    echo(a+i)
    async(){
        echo("This is from worker thread");
    }
}
```

Here loop, async are extended NPL syntax defined elsewhere.

The concept is first developed by LISP programming language in 1960s. However, despite the power of LISP, its syntax is hard to read for most programmers nowadays. NPL introduces a similar concept called `Function-Expression`, which can be mixed very well with the original NPL syntax.

The syntax of NPL's Function-Expression is `name(input,...){ ... }`

Files with `*.npl` extension support Function-Expression syntax by default. For example

```
NPL.load("(gl)script/tests/helloworld.npl")
NPL.loadstring("-- source code here", "filename_here")
```

Function-Expression User Guide

`def` expression

```
def(<name>, <params>){
    --mode:<mode>
    statements
}
```

- <name>:name of new structure to be defined, name could not be null
- <params>:parameters to be passed to defined structure multiple parameters should be seperated by comma unknown number parameters, using ...
- <mode>:mode is set at the first line of comment inside block. Mode could be strict, line and token. When different mode is set, different parsing strategy is used in defined function expression. If nothing specified, strict mode is used.

- <statements>: statements here are template code, which would be applied to final code without any change. However, one exception is +{} structure. Code inside +{} would be executed during compiling. And some default functions are provided to give users more control.

Default functions in +{}*emit(str, l)*: emit str at the line l. when no str is specified, it emit whole code chunk inside function expression block. when no l is specifiec, it emit at first line*emitline(fl, ll)*: used only in line mode. emit code chunk from line fl to ll. If no ll specified it emit from fl to end of code chunk*params(p)*: emit parameter p

UsageAfter defining a function expression, it could be used like this:

```
<name>(<params>){
     statements in specified mode
}
```

Mode

- strict: statements in strict mode are followed the rules of original npl/lua grammar
- line: statements in line mode are treated as lines, no grammar and syntax rules
- token: statements in token mode are treated as token lists, original symbols and keywords are kept, but no grammar and syntax rules

Examples

Example 1

```
def("translate", x, y, z){
    push()
    translate(+{params(x)}, +{params(y)}, +{params(z)})
    +{emit()}
    pop()
}

translate(1,2,3){
    rotate(45)
}
```

The above chunk will compiled into

```
push()
translate(1,2,3)
rotate(45)
pop()
```

Example 2

```
def("loop"){
  --mode:line
  +{local line = ast:getLines(1,1)
    local times, i = line:match("execute the code (%w+) times with (%l)")
    if not times then times="1" end
    if not i then i="i" end
       }
 for +{emit(i)}=1, +{emit(times)} do
```

```
 +{emitline(2)}
 end
 }

 loop(){execute the code 10 times with j
    print(2+j)
    print(3)
    print(4)
 }
```

The above chunk will compiled into

```
do for j=1, 10 do
print(2+j)
print(3)
print(4) end end
```

For more examples, please see our test here or dsl definition file here

CHAPTER 31

NPL C/C++ Architecture

This section covers cross-platform modules written in C/C++. These modules are exposed via NPL scripting API so that they are called via NPL.

C++ Modules

NPL Scripting Engine:

- NPL state (or NPL virtual code environment): a single NPL thread, has its own memory allocators and manages all files it load.
  - NPL state can load NPL/Lua script or C++ dll.
  - Mono state: can load C# dll.
- NPL Networking: Manage all local or remote NPL states via NPL HTTP/TCP connections.

Graphics Engine:

All GUI objects must be created in the main NPL thread, which is the same as renderer thread. 2D/3D Engine are all object oriented. All 2D objects are origanized in a parent/child tree. 3D objects are organized in a quad-tree in additional to parent/child tree.

Video/audio Renderer

A static/dynamic buffer, textures, draw 3d/2d api, fonts, etc.

- DirectX fully supported renderer.
- OpenGL renderer: a cross-platform renderer(limited functions used in our linux/android build)

PaintEngine

It is like a GDI engine used by 2D Engine for drawing 2D and simple 3d objects, like lines, rectangles, etc.

2D Engine: GUIBase is the base class to all 2d object.

- GUIRoot: root node of all GUI objects.
- GUIContainer, GUIButton, GUIText, etc

3D Engine: BaseObject is the base class to all 3D object.

- ViewportManager: manages 2D viewport into which we can render 3d or 2d objects.
- SceneObject: root node of all 3D objects. It manages all objects like adding, deleting, searching, rendering, physics, etc
- TerrainTileRoot: it is a `quadtree` container of all 3D objects for fast object searching according to their 3d locations and current camera frustum.
- CameraObject: camera frustum. several derived class like AutoCamera, etc
- MeshObject: base class static triangle mesh objects in the 3d object.
- MeshPhysicsObject: it is a static mesh with physics.
- BipedObject: it represents an animated object.
- MinisceneGraph: it is a simplified version of SceneObject, which usually manages a small number of 3d objects which can be rendered separately in to a 2D texture.
- TerrainEngine: infinitely large terrain with a heightmap, and multiple texture layers. It is rendered with dynamic level-of-detail algorithm.
- Other Scene objects:
- BlockEngine: it manages rendering of `32000x32000x256` blocks.
  - BlockRegion: manages `512x512x256` blocks, which are saved into a single file
  - ChunkColumn: `16x16x256`
  - Chunk: `16x16x16`, a static renderable object.

Physics Engine

It uses the open source Bullet physics engine for static mesh objects in the scene. Block physics is handled separately by the BlockEngine itself.

Asset Management:

Usually each static asset file is a asset entity. `AssetEntity` is the based class to all assets in the system. Assets provides data and sometimes rendering methods to be used any other 2d/3d objects. All assets are loaded asynchronously by default from either IO disk or remote network.

- TextureEntity: 2d textures
- MeshEntity: static mesh file like `.x`
- ParaXEntity: animated mesh file which can be loaded from `.x`, `.fbx`
- Audio, bmax model, database, font, etc.
- EffectFile: shader files, which can be loaded from directX `.fx`.

IO and Util

- FileManager: manages all files and search paths.
- CParaFile: manages a single file read/write.
- math: 2d/3d math libs like vectors and matrices

ParaScripting API:

- C++ –> NPL: exposing all above functions and modules to NPL scripting environment.
- See all C++ bindings in NPL scripting reference

CHAPTER 32

Core ParaEngine/NPL API

The following API is implemented in C/C++ and exposed to NPL.

For examples of using these NPL API, please refer to source code of NPL packages, like the main package

In NPL scripts, tables that begins with `Para` like `ParaIO`, `ParaUI`, `ParaScene`, `ParaEngine` are usually core C++ API. Moreover, most functions in `NPL` table is also core C++ API, like `NPL.load`, `NPL.this`, etc.

CHAPTER 33

Attribute System

Almost all C++ API objects like `ParaUIObject`, `ParaObject`, and even some global table like `ParaEngine`, expose a data interface via `ParaAttributeObject`.

The attribute system allows us to easily get or set data in core C++ object via NPL scripts, like below.

```
local attr = ParaEngine.GetAttributeObject();
local value = attr:GetField("IgnoreWindowSizeChange", false);
attr:SetField("IgnoreWindowSizeChange", not value);
```

In NPL code wiki, one can open from menu `view::object browser` to inspect all living core objects via the attribute system.

> Please note script/ide/System/Core/DOM.lua provides a handy interface to iterate over existing core objects or even pure NPL tables.

CHAPTER 34

Asset Manifest & Asynchronous Asset Loading

A graphical application usually depends on tons of assets (textures, models, etc) to process and render. It is usually not possible deploy all assets to client machine at installation time, instead assets are downloaded from the server usually on first use.

NPL/ParaEngine has built-in support for asynchronous asset loading via asset manifest system. Basically it resolves two problems:

1. Use a plain text file to look up and download assets in the background in several worker threads.
2. Most UI and 3d objects in ParaEngine provide a synchronous interface to set assets, as if they are already available. In reality, they will automatically resolves assets (also their dependencies) when those assets are available locally.

Asset Manifest Manager

When an application starts, NPLRuntime will read all `Assets_manifest*.txt` file under the root directory. Each file has following content

```
format is [relative path],md5,fileSize
```

if the name ends with .z, it is zipped. This could be 4MB uncompressed in size md5 is checksum code of the file. fileSize is the compressed file size.

```
audio/music.mp3.z,3799134715,22032
model/building/tree.dds.z,2957514200,949
model/building/tree.x.z,2551621901,816
games/tutorial.swf,1157008036,171105
```

When one of the async loader try to load an application asset(texture, model, etc), it will first search in AssetManifest using the TO-LOWER-CASED asset path, such as (model/building/tree.x). if will then search the "temp/cache/" directory for a matching file

The file matching is done by comparing the line in the asset file with the filename in the cache directory, using their md5 and size.

```
audio/music.mp3.z,3799134715,22032 matches to file 379913471522032
```

Example Usage:

```
    AssetFileEntry* pEntry = CAssetManifest::GetSingleton().GetFile("Texture/somefile.
↪dds");
    if(pEntry && pEntry->DoesFileExist())
    {
        // Load from file pEntry->GetLocalFileName();
    }
```

CHAPTER 35

ParaObject

ParaObject is the scripting proxy to a 3D object on the C++ engine. In most cases, it could be a 3d mesh, an animated character called biped, a bmax model or any custom 3D objects.

Create 3D Object

Use `CreateCharacter` to create animated character based on a ParaX asset file.

```
local player = ParaScene.CreateCharacter ("MyPlayer", ParaAsset.LoadParaX("",
↪"character/v3/Elf/Female/ElfFemale.x"), "", true, 0.35, 0, 1.0);
player:SetPosition(ParaScene.GetPlayer():GetPosition());
ParaScene.Attach(player);
```

Use `CreateMeshPhysicsObject` to create a static mesh object based on a mesh asset file

```
local asset = ParaAsset.LoadStaticMesh("","model/common/editor/z.x")
local obj = ParaScene.CreateMeshPhysicsObject("blueprint_center", asset, 1,1,1, false,
↪ "1,0,0,0,1,0,0,0,1,0,0,0");
obj:SetPosition(ParaScene.GetPlayer():GetPosition());
obj:GetAttributeObject():SetField("progress",1);
ParaScene.Attach(obj);
```

Get View Parameters

> Please note all asset are async-loaded, when asset is not loaded, object renders nothing, and following parameters may not be correct when associated mesh asset is not async-loaded.

```
local obj = ParaScene.GetObject("MyPlayer")
local params = {};
param.rotation = obj:GetRotation({})
param.scaling = obj:GetScale();
param.facing = obj:GetFacing();
param.ViewBox = obj:GetViewBox({});
local x,y,z = obj:GetViewCenter();
```

References:

More Complete API reference, please see ParaObject Reference

CHAPTER 36

System Library

All system library is contained in the main package. See here for how to install packages.

Commonly used packages

- Timer: showdoc
- Serialization
- Database
- HTTP
- Networking
- ... TODO

CHAPTER 37

Timer

- Encoding: script/ide/timer.lua

```
NPL.load("(gl)script/ide/timer.lua");

local mytimer = commonlib.Timer:new({callbackFunc = function(timer)
    commonlib.log({"ontimer", timer.id, timer.delta, timer.lastTick})
end})

-- start the timer after 0 milliseconds, and signal every 1000 millisecond
mytimer:Change(0, 1000)

-- start the timer after 1000 milliseconds, and stop it immediately.
mytimer:Change(1000, nil)

-- now kill the timer.
mytimer:Change()

-- kill all timers in the pool
commonlib.TimerManager.Clear()

-- dump timer info
commonlib.TimerManager.DumpTimerCount()

-- get the current time in millisecond. This may be faster than ParaGlobal_
↪timeGetTime() since it is updated only at rendering frame rate.
commonlib.TimerManager.GetCurrentTime();

-- one time timer
commonlib.TimerManager.SetTimeout(function()  end, 1000)
```

CHAPTER 38

Serialization, Encoding and Logging

- Serialization: script/ide/serialization.lua
- Encoding: script/ide/Encoding.lua
- Logging: script/ide/log.lua
- SHA1: script/ide/System/Encoding/sha1.lua

NPL/Lua Table Serialization

see `script/test/TestNPL.lua`

```
local o = {a=1, b="string"};

-- serialize to string
local str = commonlib.serialize_compact(o)
-- write string to log.txt
log(str);
-- string to NPL table again
local o = NPL.LoadTableFromString(str);
-- echo to output any object to log.txt
echo(o);
```

Data Formatting

```
log(string.format("succeed: test case %.3f for %s\r\n", 1/3, "hello"));
-- format is faster than string.format, but only support limited place holder like %s␣
↪and %d.
log(format("succeed: test case %d for %s\r\n", 1, "hello"));
```

Data Encoding

- Encoding: script/ide/Encoding.lua

```
commonlib.echo(NPL.EncodeURLQuery("http://www.paraengine.com", {"name1", "value1",
↪"name2", "",}))
```

UTF8 vs Default Text Encoding

Encoding is a complex topic. The rule of thumb in NPL is to use utf8 encoding wherever possible, such as in source code, XML/html/page files, UI text, network messages, etc. However, system file path must be encoded in the operating system's default encoding, which may be different from utf8, so we need to use following methods to convert between the default and utf8 encoding for a given text to be used as system file path, such as `ParaIO.open(filename)` requires `filename` to be in default encoding.

```
NPL.load("(gl)script/ide/Encoding.lua");
local Encoding = commonlib.gettable("commonlib.Encoding");
commonlib.Encoding.Utf8ToDefault(text)
commonlib.Encoding.DefaultToUtf8(text)
```

Json Encoding

- Json Encoding: script/ide/Json.lua

```
NPL.load("(gl)script/ide/Json.lua");
local t = {
["name1"] = "value1",
["name2"] = {1, false, true, 23.54, "a \021 string"},
name3 = commonlib.Json.Null()
}

local json = commonlib.Json.Encode (t)
print (json)
--> {"name1":"value1","name3":null,"name2":[1,false,true,23.54,"a \u0015 string"]}

local t = commonlib.Json.Decode(json)
print(t.name2[4])
--> 23.54

-- also consider the NPL version
local out={};
if(NPL.FromJson(json, out)) then
    commonlib.echo(out)
end
```

XML Encoding

- XML Encoding: script/ide/LuaXML.lua

```
NPL.load("(gl)script/ide/LuaXML.lua");
function TestLuaXML:test_LuaXML_CPlusPlus()
    local input = [[<paragraph justify="centered" >first child<b >bold</b>second child
↪</paragraph>]]
    local x = ParaXML.LuaXML_ParseString(input);
    assert(x[1].name == "paragraph");
end

function TestLuaXML:test_LuaXML_NPL()
    local input = [[<paragraph justify="centered" >first child<b >bold</b>second child
↪</paragraph>]]
```

```
    local xmlRoot = commonlib.XML2Lua(input)
    assert(commonlib.Lua2XmlString(xmlRoot) == input);
    log(commonlib.Lua2XmlString(xmlRoot, true))
end
```

Binary Data

To read or write binary data to or from string, we can use the file API with a special filename called `<memory>`, see below.

- To read binary string, simply call `WritingString` to write input string to a memory buffer file and then `seek(0)` and read data out in any way you like.
- To write binary string, simply call `WritingString`, `WriteBytes` or `WritingInt`, once finished, call `GetText(0,-1)` to get the final output binary string.

```
function test_MemoryFile()
    -- "<memory>" is a special name for memory file, both read/write is possible.
    local file = ParaIO.open("<memory>", "w");
    if(file:IsValid()) then
        file:WriteString("hello ");
        local nPos = file:GetFileSize();
        file:WriteString("world");
        file:WriteInt(1234);
        file:seek(nPos);
        file:WriteString("World");
        file:SetFilePointer(0, 2); -- 2 is relative to end of file
        file:WriteInt(0);
        file:WriteString("End");
        file:WriteBytes(3, {100, 0, 22});
        -- read entire binary text data back to npl string
        echo(#(file:GetText(0, -1)));
        file:close();
    end
end
```

XPath query in XML

- XPath: script/ide/XPath.lua

```
NPL.load("(gl)script/ide/XPath.lua");

local xmlDocIP = ParaXML.LuaXML_ParseFile("script/apps/Poke/IP.xml");
local xpath = "/mcml:mcml/mcml:packageList/mcml:package/";
local xpath = "//mcml:IPList/mcml:IP[@text = 'Level2_2']";
local xpath = "//mcml:IPList/mcml:IP[@version = 5]";
local xpath = "//mcml:IPList/mcml:IP[@version < 6]"; -- only supported by selectNodes2
local xpath = "//mcml:IPList/mcml:IP[@version > 4]"; -- only supported by selectNodes2
local xpath = "//mcml:IPList/mcml:IP[@version >= 5]"; -- only supported by␣
↪selectNodes2
local xpath = "//mcml:IPList/mcml:IP[@version <= 5]"; -- only supported by␣
↪selectNodes2
```

```
local xmlDocIP = ParaXML.LuaXML_ParseFile("character/v3/Pet/MGBB/mgbb.xml");
local xpath = "/mesh/shader/@index";
local xpath = "/mesh/boundingbox/@minx";
local xpath = "/mesh/submesh/@filename";
local xpath = "/mesh/submesh";

--
-- select nodes to an array table
--
local result = commonlib.XPath.selectNodes(xmlDocIP, xpath);
local result = XPath.selectNodes(xmlDocIP, xpath, nMaxResultCount); -- select at most␣
↪nMaxResultCount result

--
-- select a single node or nil
--
local node = XPath.selectNode(xmlDocIP, xpath);

--
-- iterate on all nodes.
--
for node in commonlib.XPath.eachNode(xmlDocIP, xpath) do
    commonlib.echo(node[1]);
end
```

Logging

`log.txt` may take a few seconds to be flushed to disk file.

- Logging: script/ide/log.lua

```
-- write formatted logs to log.txt
LOG.std(nil,"info", "sub_system_name", "some formated message: %s", "hello");
LOG.std(nil,"debug", "sub_system_name", {a=1, c="any table object"});
LOG.std(nil,"error", "sub_system_name", "error code here");
LOG.std(nil,"warn", "sub_system_name", "warning");
```

Log redirection and configurations: It is also possible to redirect `log.txt` to different files on startup. see NPLCommandLine

```
NPL.load("(gl)script/ide/log.lua");
commonlib.log("hello %s \n", "paraengine")
commonlib.log({"anything"})
local fromPos = commonlib.log.GetLogPos()
log(fromPos.." bababababa...\n");
local text = commonlib.log.GetLog(fromPos, nil)
log(tostring(text).." is retrieved\n")

commonlib.applog("hello paraengine"); --> ./log.txt --> 20090711 02:59:19|0|hello␣
↪paraengine|script/shell_loop.lua:23: in function FunctionName|
commonlib.applog("hello %s", "paraengine")

commonlib.servicelog("MyService", "hello paraengine"); --> ./MyService_20090711.log --
↪> 2009-07-11 10:53:27|0|hello paraengine||
commonlib.servicelog("MyService", "hello %s", "paraengine");
```

```
-- set log properties before using the log
commonlib.servicelog.GetLogger("no_append"):SetLogFile("log/no_append.log")
commonlib.servicelog.GetLogger("no_append"):SetAppendMode(false);
commonlib.servicelog.GetLogger("no_append"):SetForceFlush(true);
commonlib.servicelog("no_append", "test");

-- This will change the default logger's file position at runtime.
commonlib.servicelog.GetLogger(""):SetLogFile("log/log_2016.5.19.txt");
```

CHAPTER 39

HTTP request

http url request: script/ide/System/os/GetUrl.lua

```
-- return the content of a given url.
-- e.g.  echo(NPL.GetURL("www.paraengine.com"))
-- @param url: url string or a options table of {url=string, postfields=string, form=
↪{key=value}, headers={key=value, "line strings"}, json=bool, qs={}}
-- if .json is true, code will be decoded as json.
-- if .qs is query string table
-- if .postfields is a binary string to be passed in the request body. If this is␣
↪present, form parameter will be ignored.
-- @param callbackFunc: a function(rcode, msg, data) end, if nil, the function will␣
↪not return until result is returned(sync call).
--   `rcode` is http return code, such as 200 for success, which is same as `msg.rcode`
--   `msg` is the raw HTTP message {header, code=0, rcode=200, data}
--   `data` contains the translated response data if data format is a known format␣
↪like json
--    or it contains the binary response body from server, which is same as `msg.data`
-- @param option: mostly nil. "-I" for headers only
-- @return: return nil if callbackFunc is a function. or the string content in sync␣
↪call.
function System.os.GetUrl(url, callbackFunc, option)
end
```

```
NPL.load("(gl)script/ide/System/os/GetUrl.lua");
```

- download file or making standard request

```
System.os.GetUrl("https://github.com/LiXizhi/HourOfCode/archive/master.zip", echo);`
```

- get headers only with "-I" option.

```
System.os.GetUrl("https://github.com/LiXizhi/HourOfCode/archive/master.zip",␣
↪function(err, msg, data)  echo(msg) end, "-I");
```

- send form KV pairs with http post

```
System.os.GetUrl({url = "http://localhost:8099/ajax/console?action=getparams", form =
↪{key="value",} }, function(err, msg, data)        echo(data)  end);
```

- send multi-part binary forms with http post

```
System.os.GetUrl({url = "http://localhost:8099/ajax/console?action=printrequest",␣
↪form = {name = {file="dummy.html",    data="<html><bold>bold</bold></html>", type=
↪"text/html"}, } }, function(err, msg, data)     echo(data)  end);
```

- To send any binary data, one can use

```
System.os.GetUrl({url = "http://localhost:8099/ajax/console?action=printrequest",␣
↪headers={["content-type"]="application/json"}, postfields='{"key":"value"}' },␣
↪function(err, msg, data)       echo(data)  end);
```

- To simplify json encoding, we can send form as json string using following shortcut

```
System.os.GetUrl({url = "http://localhost:8099/ajax/console?action=getparams", json =␣
↪true, form = {key="value", key2 ={subtable="subvalue"} } }, function(err, msg,␣
↪data)      echo(data)  end);
```

HTTP PUT request:

```
System.os.GetUrl({
    method = "PUT",
    url = "http://localhost:8099/ajax/log?action=log",
    form = {filecontent = "binary string here", }
}, function(err, msg, data)     echo(data)  end);
```

HTTP DELETE request:

```
System.os.GetUrl({
    method = "DELETE",
    url = "http://localhost:8099/ajax/log?action=log",
    form = {filecontent = "binary string here", }
}, function(err, msg, data)     echo(data)  end);
```

On the server side, suppose it is NPL web server, one can get the request body using `request:GetBody()` or `request:getparams()`. The former will only include request body, while the latter contains url parameters as well.

Debugging HTTP request

In NPL Code Wiki's console window, one can test raw http request by sending a request to `/ajax/console?action=printrequest`, and then check log console for raw request content.

```
System.os.GetUrl({url = "http://localhost:8099/ajax/console?action=printrequest",␣
↪echo);
```

Local Server

Local server offers a way for you to download files or making url requests with a cache policy. Local database system is used to backup all url requests. The cached files may be in the database or in native file system.

For more information, please see localserver

what is a local server?

The LocalServer module allows a web application to cache and serve its HTTP resources locally, without a network connection.

Local Server Overview

The LocalServer module is a specialized URL cache that the web application controls. Requests for URLs in the LocalServer's cache are intercepted and served locally from the user's disk.

Resource stores

A resource store is a container of URLs. Using the LocalServer module, applications can create any number of resource stores, and a resource store can contain any number of URLs.

There are two types of resource stores: - ResourceStore - for capturing ad-hoc URLs using NPL. The ResourceStore allows an application to capture user data files that need to be addressed with a URL, such as a PDF file or an image. - ManagedResourceStore - for capturing a related set of URLs that are declared in a manifest file, and are updated automatically. The ManagedResourceStore allows the set of resources needed to run a web application to be captured. For both types of stores, the set of URLs captured is explicitly controlled by the web application.

Architecture & Implementation Notes

all sql database manipulation functions are exposed via WebCacheDB, whose implementation is split in WebCacheDB* files. localserver is the based class for two servers: ResourceStore and ManagedResourceStore.

Using Local server as a local database

One can use local server as a simple (name, value) pair database with cache_policy functions.

To query a database entry call below, here we will use web service store

```
    NPL.load("(gl)script/ide/System/localserver/factory.lua");
    local ls = System.localserver.CreateStore(nil, 2);
    if(not ls) then
        return
    end
    cache_policy = cache_policy or System.localserver.CachePolicy:new("access plus 1␣
↪week");

    local url = System.localserver.UrlHelper.WS_to_REST(fakeurl_query_miniprofile,
↪{JID=JID}, {"JID"});
    local item = ls:GetItem(url)
    if(item and item.entry and item.payload and not cache_policy:IsExpired(item.
↪payload.creation_date)) then
        -- NOTE:item.payload.data is always a string, one may deserialize from it to␣
↪obtain table object.
        local profile = item.payload.data;
        if(type(callbackFunc) == "function") then
            callbackFunc(JID, profile);
        end
    else

    end
```

To add(update) a database entry call below

```
    NPL.load("(gl)script/ide/System/localserver/factory.lua");
    local ls = System.localserver.CreateStore(nil, 2);
    if(not ls) then
        return
    end
    -- make url
    local url = System.localserver.UrlHelper.WS_to_REST(fakeurl_query_miniprofile,
↪{JID=JID}, {"JID"});

    -- make entry
    local item = {
        entry = System.localserver.WebCacheDB.EntryInfo:new({
            url = url,
        }),
        payload = System.localserver.WebCacheDB.PayloadInfo:new({
            status_code = System.localserver.HttpConstants.HTTP_OK,
            data = msg.profile,
        }),
    }
    -- save to database entry
    local res = ls:PutItem(item)
    if(res) then
        log("ls put JID mini profile for "..url.."\n")
    else
        log("warning: failed saving JID profile item to local server.\n")
    end
```

Lazy writing

For the URL history, this transaction commit overhead is unacceptably high(0.05s for the most simple write commit). On some systems, the cost of committing a new page to the history database was as high as downloading the entire page and rendering the page to the screen. As a result, ParaEngine's localserver has implemented a lazy sync system.

Please see https://developer.mozilla.org/en/Storage/Performance, for a reference

Localserver has relaxed the ACID requirements in order to speed up commits. In particular, we have dropped durability. This means that when a commit returns, you are not guaranteed that the commit has gone through. If the power goes out right away, that commit may (or may not) be lost. However, we still support the other (ACI) requirements. This means that the database will not get corrupted. If the power goes out immediately after a commit, the transaction will be like it was rolled back: the database will still be in a consistent state.

Send Email Via SMTP

It is not trivial to make a SMTP email server. However, it is fairly easy to send an email via an existing email server. One can do it manually via `System.os.GetUrl` with proper options, or we write the following handy function for you.

```
System.os.SendEmail({
    url="smtp://smtp.exmail.qq.com",
    username="lixizhi@paraengine.com", password="XXXXX",
    -- ca_info = "/path/to/certificate.pem",
```

```
    from="lixizhi@paraengine.com", to="lixizhi@yeah.net", cc="xizhi.li@gmail.com",
    subject = "title here",
    body = "any body context here. can be very long",
}, function(err, msg) echo(msg) end);
```

Here is what I receive in my mail box:

CHAPTER 40

Files API

Search Files

- Files: script/ide/Files.lua

```
NPL.load("(gl)script/ide/Files.lua");
local result = commonlib.Files.Find({}, "model/test", 0, 500, function(item)
    local ext = commonlib.Files.GetFileExtension(item.filename);
    if(ext) then
        return (ext == "x") or (ext == "dds")
    end
end)

-- search zip files using perl regular expression. like ":^xyz\\s+.*blah$"
local result = commonlib.Files.Find({}, "model/test", 0, 500, ":.*", "*.zip")

-- using lua file system
local lfs = commonlib.Files.GetLuaFileSystem();
echo(lfs.attributes("config/config.txt", "mode"))
```

Read/Write Binary Files

```
    local file = ParaIO.open("temp/binaryfile.bin", "w");
    if(file:IsValid()) then
        local data = "binary\0\0\0\0file";
        file:WriteString(data, #data);
        -- write 32 bits int
        file:WriteUInt(0xffffffff);
        file:WriteInt(-1);
        -- write float
        file:WriteFloat(-3.14);
        -- write double (precision is limited by lua double)
        file:WriteDouble(-3.1415926535897926);
        -- write 16bits word
        file:WriteWord(0xff00);
        -- write 16bits short integer
        file:WriteShort(-1);
        file:WriteBytes(3, {255, 0, 255});
        file:close();
```

```
        -- testing by reading file content back
        local file = ParaIO.open("temp/binaryfile.bin", "r");
        if(file:IsValid()) then
            -- test reading binary string without increasing the file cursor
            assert(file:GetText(0, #data) == data);
            file:seekRelative(#data);
            assert(file:getpos() == #data);
            file:seek(0);
            -- test reading binary string
            assert(file:ReadString(#data) == data);
            assert(file:ReadUInt() == 0xffffffff);
            assert(file:ReadInt() == -1);
            assert(math.abs(file:ReadFloat() - (-3.14)) < 0.000001);
            assert(file:ReadDouble() == -3.1415926535897926);
            assert(file:ReadWord() == 0xff00);
            assert(file:ReadShort() == -1);
            local o = {};
            file:ReadBytes(3, o);
            assert(o[1] == 255 and o[2] == 0 and o[3] == 255);
            file:seek(0);
            assert(file:ReadString(8) == "binary\0\0");
            file:close();
        end
    end
```

In-Memory File And Binary Buffer

To write binary data to string, we can use the file API with a special filename called `<memory>`, see below

```
function test_MemoryFile()
    -- "<memory>" is a special name for memory file, both read/write is possible.
    local file = ParaIO.open("<memory>", "w");
    if(file:IsValid()) then
        file:WriteString("hello ");
        local nPos = file:GetFileSize();
        file:WriteString("world");
        file:WriteInt(1234);
        file:seek(nPos);
        file:WriteString("World");
        file:SetFilePointer(0, 2); -- 2 is relative to end of file
        file:WriteInt(0);
        file:WriteString("End");
        file:WriteBytes(3, {100, 0, 22});
        -- read entire binary text data back to npl string
        echo(#(file:GetText(0, -1)));
        file:close();
    end
end
```

Read/Write Files

see `NPL.load("(gl)script/test/ParaIO_test.lua");`

```
-- tested on 2007.1, LiXizhi
local function ParaIO_FileTest()

    -- file write
    log("testing file write...\r\n")

    local file = ParaIO.open("temp/iotest.txt", "w");
    file:WriteString("test\r\n");
    file:WriteString("test\r\n");
    file:close();

    -- file read
    log("testing file read...\r\n")

    local file = ParaIO.open("temp/iotest.txt", "r");
    log(tostring(file:readline()));
    log(tostring(file:readline()));
    log(tostring(file:readline()));
    file:close();

end

-- tested on 2007.6.7, LiXizhi
local function ParaIO_ZipFileTest()
    local writer = ParaIO.CreateZip("d:\\simple.zip","");
    writer:ZipAdd("temp/file1.ini", "d:\\file1.ini");
    writer:ZipAdd("temp/file2.ini", "d:\\file2.ini");
    writer:ZipAddFolder("temp");
    writer:AddDirectory("worlds/", "d:/temp/*.", 4);
    writer:AddDirectory("worlds/", "d:/worlds/*.*", 2);
    writer:close();
end

-- tested on 2007.6.7, LiXizhi
local function ParaIO_SearchZipContentTest()
    -- test case 1
    log("test case 1\n");
    local search_result = ParaIO.SearchFiles("","*.", "d:\\simple.zip", 0, 10000, 0);
    local nCount = search_result:GetNumOfResult();
    local i;
    for i = 0, nCount-1 do
        log(search_result:GetItem(i).."\n");
    end
    search_result:Release();
    -- test case 2
    log("test case 2\n");
    local search_result = ParaIO.SearchFiles("","*.ini", "d:\\simple.zip", 0, 10000,␣
↪0);
    local nCount = search_result:GetNumOfResult();
    local i;
    for i = 0, nCount-1 do
        log(search_result:GetItem(i).."\n");
    end
    search_result:Release();
    -- test case 3
    log("test case 3\n");
    local search_result = ParaIO.SearchFiles("","temp/*.", "d:\\simple.zip", 0, 10000,
↪ 0);
```

```
    local nCount = search_result:GetNumOfResult();
    local i;
    for i = 0, nCount-1 do
        log(search_result:GetItem(i)..[[\n]]);
    end
    search_result:Release();
    -- test case 4
    log("test case 4\n");
    local search_result = ParaIO.SearchFiles("temp/","*.*", "d:\\simple.zip", 0,␣
↪10000, 0);
    local nCount = search_result:GetNumOfResult();
    local i;
    for i = 0, nCount-1 do
        log(search_result:GetItem(i)..[[\n]]);
    end
    search_result:Release();
    -- test case 5
    log("test case 5\n");
    local search_result = ParaIO.SearchFiles("","temp/*.*", "d:\\simple.zip", 0,␣
↪10000, 0);
    local nCount = search_result:GetNumOfResult();
    local i;
    for i = 0, nCount-1 do
        log(search_result:GetItem(i)..[[\n]]);
    end
    search_result:Release();
end

local function ParaIO_SearchPathTest()
    ParaIO.CreateDirectory("npl_packages/test/")
    local file = ParaIO.open("npl_packages/test/test_searchpath.lua", "w");
    file:WriteString("echo('from test_searchpath.lua')")
    file:close();

    ParaIO.AddSearchPath("npl_packages/test");
    ParaIO.AddSearchPath("npl_packages/test/"); -- same as above, check for duplicate

    assert(ParaIO.DoesFileExist("test_searchpath.lua"));

    ParaIO.RemoveSearchPath("npl_packages/test");

    assert(not ParaIO.DoesFileExist("test_searchpath.lua"));

    -- ParaIO.AddSearchPath("npl_packages/test/");
    -- this is another way of ParaIO.AddSearchPath, except that it will check for␣
↪folder existence.
    -- in a number of locations.
    NPL.load("npl_packages/test/");

    -- test standard open api
    local file = ParaIO.open("test_searchpath.lua", "r");
    if(file:IsValid()) then
        log(tostring(file:readline()));
    else
        log("not found\n");
    end
    file:close();
```

```
    -- test script file
    NPL.load("(gl)test_searchpath.lua");

    ParaIO.ClearAllSearchPath();

    assert(not ParaIO.DoesFileExist("test_searchpath.lua"));
end

-- TODO: test passed on 2008.4.20, LiXizhi
function ParaIO_PathReplaceable()
    ParaIO.AddPathVariable("WORLD", "worlds/MyWorld")
    if(ParaIO.AddPathVariable("userid", "temp/LIXIZHI_PARAENGINE")) then
        local fullpath;
        commonlib.echo("test simple");
        fullpath = ParaIO.DecodePath("%WORLD%/%userid%/filename");
        commonlib.echo(fullpath);
        fullpath = ParaIO.EncodePath(fullpath)
        commonlib.echo(fullpath);

        commonlib.echo("test encoding with a specified variables");
        fullpath = ParaIO.DecodePath("%WORLD%/%userid%/filename");
        commonlib.echo(fullpath);
        commonlib.echo(ParaIO.EncodePath(fullpath, "WORLD"));
        commonlib.echo(ParaIO.EncodePath(fullpath, "WORLD, userid"));

        commonlib.echo("test encoding with inline path");
        fullpath = ParaIO.DecodePath("%WORLD%/%userid%_filename");
        commonlib.echo(fullpath);
        fullpath = ParaIO.EncodePath(fullpath)
        commonlib.echo(fullpath);


        commonlib.echo("test nested");
        fullpath = ParaIO.DecodePath("%userid%/filename/%userid%/nestedtest");
        commonlib.echo(fullpath);
        fullpath = ParaIO.EncodePath(fullpath)
        commonlib.echo(fullpath);

        commonlib.echo("test remove");
        if(ParaIO.AddPathVariable("userid", nil)) then
            fullpath = ParaIO.DecodePath("%userid%/filename");
            commonlib.echo(fullpath);
            fullpath = ParaIO.EncodePath(fullpath)
            commonlib.echo(fullpath);
        end

        commonlib.echo("test full path");
        fullpath = ParaIO.DecodePath("NormalPath/filename");
        commonlib.echo(fullpath);
        fullpath = ParaIO.EncodePath(fullpath)
        commonlib.echo(fullpath);
    end
end

function ParaIO_SearchFiles_reg_expr()
    -- test case 1
    local search_result = ParaIO.SearchFiles("script/ide/",":.*", "*.zip", 2, 10000,␣
↪0);
```

```
    local nCount = search_result:GetNumOfResult();
    local i;
    for i = 0, nCount-1 do
        log(search_result:GetItem(i).."\n");
    end
    search_result:Release();
end

function test_excel_doc_reader()
    NPL.load("(gl)script/ide/Document/ExcelDocReader.lua");
    local ExcelDocReader = commonlib.gettable("commonlib.io.ExcelDocReader");
    local reader = ExcelDocReader:new();

    -- schema is optional, which can change the row's keyname to the defined value.
    reader:SetSchema({
        [1] = {name="npcid", type="number"},
        [2] = {name="superclass", validate_func=function(value)  return value or
↪"menu1"; end },
        [3] = {name="class", validate_func=function(value)  return value or "normal";␣
↪end },
        [4] = {name="class_name", validate_func=function(value)  return value or "";␣
↪end },
        [5] = {name="gsid", type="number" },
        [6] = {name="exid", type="number" },
        [7] = {name="money_list", },
    })
    -- read from the second row
    if(reader:LoadFile("config/Aries/NPCShop/npcshop.xml", 2)) then
        local rows = reader:GetRows();
        echo(rows);
    end



    NPL.load("(gl)script/ide/Document/ExcelDocReader.lua");
    local ExcelDocReader = commonlib.gettable("commonlib.io.ExcelDocReader");
    local reader = ExcelDocReader:new();

    -- schema is optional, which can change the row's keyname to the defined value.
    local function card_and_level_func(value)
        if(value) then
            local level, card_gsid = value:match("^(%d+):(%d+)");
            return {level=level, card_gsid=card_gsid};
        end
    end
    reader:SetSchema({
        {name="gsid", type="number"},
        {name="displayname"},
        {name="max_level", type="number" },
        {name="hp", type="number" },
        {name="attack", type="number" },
        {name="defense", type="number" },
        {name="powerpips_rate", type="number" },
        {name="accuracy", type="number" },
        {name="critical_attack", type="number" },
        {name="critical_block", type="number" },
        {name="card1",  validate_func= card_and_level_func},
        {name="card2",  validate_func= card_and_level_func},
```

```
        {name="card3",  validate_func= card_and_level_func},
        {name="card4",  validate_func= card_and_level_func},
        {name="card5",  validate_func= card_and_level_func},
        {name="card6",  validate_func= card_and_level_func},
        {name="card7",  validate_func= card_and_level_func},
        {name="card8",  validate_func= card_and_level_func},
    })
    -- read from the second row
    if(reader:LoadFile("config/Aries/Others/combatpet_levels.excel.teen.xml", 2)) then
        local rows = reader:GetRows();
        log(commonlib.serialize(rows, true));
    end
end
```

Delete Files

See ParaIO Reference for more functions

```
ParaIO.DeleteFile("temp/*.*");
```

Compress and Decompress with Zlib and Gzip

- Files: script/test/TestNPL.lua

```
-- compress/decompress test
function TestNPL.Compress()
    -- using gzip
    local content = "abc";
    local dataIO = {content=content, method="gzip"};
    if(NPL.Compress(dataIO)) then
        echo(dataIO);
        if(dataIO.result) then
            dataIO.content = dataIO.result; dataIO.result = nil;
            if(NPL.Decompress(dataIO)) then
                echo(dataIO);
                assert(dataIO.result == content);
            end
        end
    end

    -- using zlib and deflate
    local content = "abc";
    local dataIO = {content=content, method="zlib", windowBits=-15, level=3};
    if(NPL.Compress(dataIO)) then
        echo(dataIO);
        if(dataIO.result) then
            dataIO.content = dataIO.result; dataIO.result = nil;
            if(NPL.Decompress(dataIO)) then
                echo(dataIO);
                assert(dataIO.result == content);
            end
        end
```

```
    end
end
```

Running Shell Commands

- source: script/ide/System/os/run.lua

To run external command lines via windows batch or linux bash shell in either `synchronous` or `asynchronous` mode.

```
NPL.load("(gl)script/ide/System/os/run.lua");
if(System.os.GetPlatform()=="win32") then
    -- any lines of windows batch commands
    echo(System.os("dir *.exe \n svn info"));
    -- this will popup confirmation window, so there is no way to get its result.
    System.os.runAsAdmin('reg add "HKCR\\paracraft" /ve /d "URL:paracraft" /f');
else
    -- any lines of linux bash shell commands
    echo(System.os.run("ls -al | grep total\ngit | grep commit"));
end
-- async run command in worker thread
for i=1, 10 do
    System.os.runAsync("echo hello", function(err, result)   echo(result)  end);
end
echo("waiting run async reply ...")
```

please note:

- windows and linux have different default shell program. One may need to use `System.os.GetPlatform()=="win32"` to target code to a given platform.
- We can write very long multi-line commands. Internally a temporary shell script file is created and executed with IO redirection.
- Sometimes, we may prefer async API to prevent stalling the calling thread, such as NPL web server invoking some backend programs for image processing. The async API pushes a message to a processor queue and returns immediately, one can use one or more NPL threads to process any number of queued shell commands. For more details, please see doc in `System.os.runAsync` source file.
- You need to have permissions to run these scripts. Under window 10 or above, we provide `System.os.runAdmin` to automatically popup confirmation dialog to run as administrator. Under linux, there is nothing we can do, you must give execute permission to `./temp` folder. This is the case for `selinux` core.

Reading Image File

```
function test_reading_image_file()
    -- reading binary image file
    -- png, jpg format are supported.
    local filename = "Texture/alphadot.png";
    local file = ParaIO.open(filename, "image");
    if(file:IsValid()) then
        local ver = file:ReadInt();
        local width = file:ReadInt();
```

```
        local height = file:ReadInt();
        -- how many bytes per pixel, usually 1, 3 or 4
        local bytesPerPixel = file:ReadInt();
        echo({ver, width=width, height = height, bytesPerPixel = bytesPerPixel})
        local pixel = {};
        for y=1, height do
            for x=1, width do
                pixel = file:ReadBytes(bytesPerPixel, pixel);
                echo({x, y, rgb=pixel})
            end
        end
        file:close();
    end
end
```

CHAPTER 41

Mouse and Key Input

This section is about handling mouse and key events in a window application.

Low level event handler

At the lowest level of NPL, mouse and keyboard event handlers can be registered to `ParaUIObject` and the global `ParaScene`. The ParaScene is a global singleton to handle all events not handled by any GUI objects. Please note, there is a another `AutoCameraController` which is enabled by default to handle mouse and key events before passing to ParaScene. So the order of event filtering in NPLRuntime is like below

- GUI events: controls that have focus always get event first
- `AutoCameraController`: optionally enabled for handling basic `player control`, such as right click to rotate the view, arrow keys to move the main player. Please note, this can be disabled if one wants to handle everything in NPL script. Trust me, it can cost you over 2000 lines of code if you do it manually.
- ParaScene: finally mouse and key events are handled by the 3d scene.

It is NOT recommended to use low level NPL api directly, see next section.

Use NPL libraries to handle event

2D events

For 2D GUI events, one can handle via window controls or MCML page. There is also a mcml v1 tag called `<pe:hotkey>` for handling simple key event when window is visible.

3D Scene Context

For 3D or global mouse/key, one should use SceneContext.

At most one scene context can be selected at any time. Once selected, the scene context will receive all key/mouse events in the 3D scene. One can derive from this class to write and switch to your own scene event handlers.

The computational model for global key/mouse event in C++ Engine is:

```
key/mouse input--> 2D --> (optional auto camera) --> 3D scene --> script handlers
```

This class simplified and modified above model with following object oriented model:

```
    key/mouse input--> 2D --> one of SceneContext object--> Manipulator container -->␣
↪ Manipulators --> (optional auto camera manipulator)
```

Please note that both model can coexist, however SceneContext is now the recommended way to handle any scene event. SceneContext hide all dirty work of hooking into the old C++ engine callback interface, and offers more user friendly way of event handling.

Virtual functions:

```
    mousePressEvent(event)
    mouseMoveEvent
    mouseReleaseEvent
    mouseWheelEvent
    keyReleaseEvent
    keyPressEvent
    OnSelect()
    OnUnselect()
```

use the lib:

```
NPL.load("(gl)script/ide/System/Core/SceneContext.lua");
local MySceneContext = commonlib.inherit(commonlib.gettable("System.Core.SceneContext
↪"), commonlib.gettable("System.Core.MySceneContext"));
function MySceneContext:ctor()
    self:EnableAutoCamera(true);
end

function MySceneContext:mouseReleaseEvent(event)
    _guihelper.MessageBox("clicked")
end

-- method 1:
local sContext = MySceneContext:new():Register("MyDefaultSceneContext");
sContext:activate();
-- method 2:
MySceneContext:CreateGetInstance("MyDefaultSceneContext"):activate();
```

Scene Context Switch

Context is usually associated with a tool item, such as when user click to use the tool, its context is activated and once user deselect it, the context is switched back to a default one.

Scene context allows one to write modular code for each tool items with complex mouse/key input without affecting each other. It is common for an application to derive all of its context from a base context to support shared global key events, etc.

For an example of using context, please see Paracraft's context folder.

> See also here for how to replace the default context in paracraft's mod interface with filters.

Manipulators

Scene context can contain manipulators. Manipulator is a module to manipulate complex scene objects, like the one below.

Manipulator is the base class used for creating user-defined manipulators. A manipulator can be connected to a depend node instead of updating a node attribute directly call AddValue() in constructor if one wants to define a custom manipulator property(plug) that can be easily binded with dependent node's plug.

Manipulator can also draw complex 3D overlay objects with 3d picking support. In above picture, the three curves are drawn by the Rotate manipulator.

CHAPTER 42

Overview

To write modular code, a system may expose its interface in the form of virtual functions, events or filters.

- `virtual functions`: allows a derived class to hook the input and output of a single function in the base class.
- `events`: feeds input to external functions when certain things happened. Events only hooks the input without providing an output.
- `filters`: allows external functions to form input-output chains to modify data at any point of execution.

Use whatever patterns to allow external modifications to the input-output of your modular code.

Filters

Filters is a design pattern of input-output chains. Please see script/ide/System/Core/Filters.lua for detailed usage.

You can simply apply a named filters at any point of execution in your code to allow other users to modify your data.

For example, one can turn `data = data;` into an equivalent filter, like below

```
data = GameLogic.GetFilters():apply_filters("my_data", data, some_parameters);
```

The above code does nothing until some other modules `add_filters` to `"my_data"`.

Each filter function takes the output of the previous filter function as its first parameter, all other input parameters are shared by all filter functions, like below

```
F1(input, ...)-->F2(F1_output, ...)-->F3(F2_output, ...)-->F3_output
```

Filters is the recommended way for plugin/app developers to extend or modify paracraft.

> Click here to see all Paracraft Filters

CHAPTER 43

Localization

There are some helper class for you to implement localization. For best practices, please see example in paracraft package, which uses `poedit` to edit and store localization strings with multi-languages.

- Helper class for translation table:
- paracraft examples:

Overview

UTF8 encoding should be used where ever possible in your source code or mcml files. Everything returned from NPL is also UTF8 encoded, except for native file path.

Follow following steps:

- Use this class to create a global table `L` to look up for localized text with a primary key.
- When your application start, populate `table L` with data from your localization files according to current language setting.
- In your script code or mcml UI page files, replace any text "XXX" with `L"XXX"`
- Re-Run `Poedit` to scan for all source files and update the localization database file.
- Inform your translator to translate the `*.po` files into multiple languages and generate `*.mo` files. Or you may just use google translate or other services.
- Iterate above three steps when you insert new text in your source code.

  Always use

```
-- Right way
local text = format(L"Some text %d times", times)
```

instead of

```
-- BAD & Wrong Way!!!
local text = L"Some text "..times..L" times"
```

for better translation because different language have different syntax.

Localization GetText Tools

To store your language strings, you can use plain table in script or use a third-party tool like Poedit. We have created `Poedit` plugin to support NPL code, download the NPLgettext tool here.

In paracraft, there is a command called `/poedit` which does the `gettext` job automatically.

CHAPTER 44

User Interface

There are two low-level ways to draw 2d graphical objects.

- One is via creating ParaUIObject controls (like buttons, containers, editbox). These controls are managed in C++ and created in NPL scripts.
- The other is creating owner draw ParaUIObject, and do all the drawings with Painting API in NPL scripts, such as draw rect, lines, text, etc.

Moreover, there are a number of high-level graphical libraries written in NPL which allows you to create 2D user interface more easily.

- IDE controls is NPL wrapper of the low-level C++ API. It provides more versatile controls than the raw ParaUIObjects.
  - One old implementation is based on ParaUIObject controls.
  - One new implementation is based on Painting API, in `System.Window.UIElement` namespace.
- MCML/NPL is a HTML/JS like mark up language to create user interface.
  - One old implementation is based on ParaUIObject controls.
  - One new implementation is based on Painting API, in `System.Window.mcml` namespace.

Further Reading

CHAPTER 45

Drawing With 2D API

There are two low-level ways to draw 2d graphical objects.

- One is via creating ParaUIObject controls (like buttons, containers, editbox). These controls are managed in C++ and created in NPL scripts.
- The other is creating owner draw ParaUIObject, and do all the drawings with Painting API in NPL scripts, such as draw rect, lines, text, etc.

Moreover, there is a number of high-level graphical libraries written in NPL which allows you to create 2D user interface more easily.

- IDE controls is NPL wrapper of the low-level C++ API. It provides more versatile controls than the raw ParaUIObjects.
  - One old implementation is based on ParaUIObject controls.
  - One new implementation is based on Painting API, in `System.Window.UIElement` namespace.
- MCML/NPL is a HTML/JS like mark up language to create user interface.
  - One old implementation is based on ParaUIObject controls.
  - One new implementation is based on Painting API, in `System.Window.mcml` namespace.

  This article is only about low-level drawing API in C++ side, which are exposed to NPL. However, one should always use MCML or Windows.UIElement instead of following raw API.

button

Creating a button with a texture background and text. The root element is a container see next section. Please place a png texture `Texture/alphadot.png` at the working directory.

```
NPL.load("(gl)script/ide/System/System.lua");

local clickCount = 1;
local function CreateUI()
   local _this = ParaUI.CreateUIObject("button", "MyBtn", "_lt", 10, 10, 64, 22);
   _this.text= "text";
   _this.background = "Texture/alphadot.png";
   _this:SetScript("onclick", function(obj)
      obj.text = "clicked "..clickCount;
      clickCount = clickCount + 1;
   end)
   _this:AttachToRoot();
```

```
end
CreateUI();
```

container

Create a container and a child button inside it.

```
    local _parent, _this;
    _this = ParaUI.CreateUIObject("container", "MyContainer", "_lt", 10, 110, 200,␣
↪64);
    _this:AttachToRoot();
    _this.background = "Texture/alphadot.png";
    _parent = _this;

    -- create a child
    _this = ParaUI.CreateUIObject("button", "b", "_rt", -10-32, 10, 32, 22);
    _this.text= "X"
    _this.background = "";
    _parent:AddChild(_this);
```

text and editbox

```

```

System Fonts

In ParaEngine/NPL, we support loading fonts from `*.ttf` files. One can install custom font files at `fonts/*.ttf`, and use them with filename, such as `Verdana;bold`. Please note, all controls in ParaEngine uses "System" font by default. "System" font maps to the system font used by the current computer by default. However, one can change it to a different custom font installed in `fonts/*.ttf`. Suppose you have a font file at `fonts/ParaEngineThaiFont.ttf`, you can map default System font to it by calling.

```
Config.AppendTextValue("GUI_font_mapping","System");
Config.AppendTextValue("GUI_font_mapping","ParaEngineThaiFont");
```

It is a pair of function calls, the first is name, the second is value. We recommend doing it in `script/config.lua`, it is loaded before any UI control is created. Please see that file for details.

Please note, for each font with different size, weight, and name, we will create a different internal temporary image file for rendering. So it is recommended to use just a few fonts in your application.

CHAPTER 46

2D GUI Windows

A `System.Windows.Window` is the primary container for 2D GUI. There are multiple ways to create windows. The most common way is to use `mcml` markup language. The programing model is like writing `HTML/js` web page.

Quick Sample

First you need to create a html file, such as `source/HelloWorldMCML/mcml_window.html`.

```
<pe:mcml>
<script refresh="false" type="text/npl" src="mcml_window.lua"><![CDATA[
    function OnClickOK()
        local content = Page:GetValue("content") or "";
        _guihelper.MessageBox(":"..content);
    end
]]></script>
<div style="color:#33ff33;background-color:#808080">
    <div style="margin:10px;">
        Hello World from HTML page!
    </div>
    <div style="margin:10px;">
        <input type="text" name="content" style="width:200px;height:25px;" />
        <input type="button" name="ok" value="" onclick="OnClickOK"/>
    </div>
</div>
</pe:mcml>
```

To create and show the window using the html file, we simply do following code.

```
    NPL.load("(gl)script/ide/System/Windows/Window.lua");
    local Window = commonlib.gettable("System.Windows.Window")
    local window = Window:new();
    window:Show({
        url="source/HelloWorldMCML/mcml_window.html",
        alignment="_lt", left = 300, top = 100, width = 300, height = 400,
    });
```

Window Alignment

Notice the alignment parameter specify the relative position to its parent or the native screen window.

```
        * @param alignment: can be one of the following strings or nil or left out␣
↪entirely:
        *   - "_lt" align to left top of the screen
        *   - "_lb" align to left bottom of the screen
        *   - "_ct" align to center of the screen
        *   - "_ctt": align to center top of the screen
        *   - "_ctb": align to center bottom of the screen
        *   - "_ctl": align to center left of the screen
        *   - "_ctr": align to center right of the screen
        *   - "_rt" align to right top of the screen
        *   - "_rb" align to right bottom of the screen
        *   - "_mt": align to middle top
        *   - "_ml": align to middle left
        *   - "_mr": align to middle right
        *   - "_mb": align to middle bottom
        *   - "_fi": align to left top and right bottom. This is like fill in the␣
↪parent window.
        *
        *   the layout is given below:
        *   _lt _mt _rt
        *   _ml _ct _mr
        *   _lb _mb _rb
```

`left,top,width,height` have different meanings for different alignment type. The most simple one is "_lt" which means left top alignment.

- center alignment:
  - `alignment="_ct",left = 0,top = -100,width = 300,height = 400` will display window at center. Normally, if you want to center a window with given `width` and `height`, one should use: `left = -width/2,top = -height/2,width,height`. The left, top means relative to the center of the parent window. "_ctt" means center top.
- middle alignment:
  - "_mt" middle top: it means that the pixels relative to left and right border of the parent window should be `fixed`.In other words, if the parent resizes, the width of the window also resize.
  - `_mt`: x is coordinate from the left. y is coordinate from the top, width is the coordinate from the right and height is the height
  - `_mb`: x is coordinate from the left. y is coordinate from the bottom, width is the coordinate from the right and height is the height
  - `_ml`: x is coordinate from the left. y is coordinate from the top, width is the width and height is the coordinate from the bottom
  - `_mr`: x is coordinate from the right. y is coordinate from the top, width is the width and height is the coordinate from the bottom

MCML V1 vs V2

There two implementations of MCML: `v1` and `v2`. The example above is given by v2, which is the preferred implementation. However, v1 is mature and stable implementation, v2 is new and currently does not have as many build-in controls as v1. We are still working on v2 and hope it can be 100% compatible and powerful with v1. To launch the same mcml page with v1, one can use

```
    NPL.load("(gl)script/kids/3DMapSystemApp/mcml/PageCtrl.lua");
    local page = System.mcml.PageCtrl:new({url="source/HelloWorldMCML/mcml_window.html
↪"});
    page:Create("testpage", nil, "_lt", 0, 0, 300, 400);
```

Difference between v1 and v2

- v1 uses NPL's build-in GUI objects written in C++, i.e. ParaUIObject like `button`,`container`,`editbox`. Their implementation is hidden from NPL scripts and user IO is exposed to NPL script via callback functions.
- v2 uses NPL script to draw all the GUI objects using 3d-triangle drawing API and handles user input in NPL script. Therefore the user has complete control over the look of their control in NPL script. One can read the source code of v2 in main package.

UI Elements (Controls)

In mcml v2 implementation, mcml page are actually translated to UI Elements at runtime. UI Element contains a collection of GUI controls written completely in NPL. They all derive from UIElement, which contains virtual functions to paint the control and handle user IO event.

The following are buildin-in UI elements or controls:

Creating GUI Without MCML

In most cases, we should use mcml to create GUI, however, there are places when you want to create your custom control or custom mcml tag, you will then need to write your own control.

Following are some examples:

```
function test_Windows:TestCreateWindow()

    -- create the native window
    local window = Window:new();

    -- test UI element
    local elem = UIElement:new():init(window);
    elem:SetBackgroundColor("#0000ff");
    elem:setGeometry(10,0,64,32);

    -- test create rectangle
    local rcRect = Rectangle:new():init(window);
    rcRect:SetBackgroundColor("#ff0000");
    rcRect:setGeometry(10,32,64,32);

    -- test Button
    local btn = Button:new():init(window);
    btn:SetBackgroundColor("#00ff00");
    btn:setGeometry(10,64,64,32);
    btn:Connect("clicked", function (event)
        _guihelper.MessageBox("you clicked me");
    end)
    btn:Connect("released", function(event)
        _guihelper.MessageBox("mouse up");
    end)
```

```
    -- show the window natively
    window:Show("my_window", nil, "_mt", 0,0, 200, 200);
end

function test_Windows:TestMouseEnterLeaveEvents()
    -- create the native window
    local window = Window:new();
    window.mouseEnterEvent = function(self, event)
        Application:postEvent(self, System.Core.LogEvent:new({"window enter", event:
↪localPos()}));
    end
    window.mouseLeaveEvent = function(self, event)
        Application:postEvent(self, System.Core.LogEvent:new({"window leave"}));
    end

    -- Parent1
    local elem = UIElement:new():init(window);
    elem:SetBackgroundColor("#0000ff");
    elem:setGeometry(10,0,64,64);
    elem.mouseEnterEvent = function(self, event)
        Application:postEvent(self, System.Core.LogEvent:new({"parent1 enter", event:
↪localPos()}));
    end
    elem.mouseLeaveEvent = function(self, event)
        Application:postEvent(self, System.Core.LogEvent:new({"parent1 leave"}));
    end

    -- Parent1:Button1
    local btn = Button:new():init(elem);
    btn:SetBackgroundColor("#ff0000");
    btn:setGeometry(0,0,64,32);
    btn.mouseEnterEvent = function(self, event)
        Application:postEvent(self, System.Core.LogEvent:new({"btn1 enter", event:
↪localPos()}));
    end
    btn.mouseLeaveEvent = function(self, event)
        Application:postEvent(self, System.Core.LogEvent:new({"btn1 leave"}));
    end

    -- Button2
    local btn = Button:new():init(window);
    btn:SetBackgroundColor("#00ff00");
    btn:setGeometry(10,64,64,32);
    btn.mouseEnterEvent = function(self, event)
        Application:postEvent(self, System.Core.LogEvent:new({"btn2 enter", event:
↪localPos()}));
    end
    btn.mouseLeaveEvent = function(self, event)
        Application:postEvent(self, System.Core.LogEvent:new({"btn2 leave"}));
    end

    -- show the window natively
    window:Show("my_window1", nil, "_mt", 0,200, 200, 200);
end

function test_Windows:TestEditbox()
```

```
    -- create the native window
    local window = Window:new();

    -- test UI element
    local elem = EditBox:new():init(window);
    elem:setGeometry(60,30,64,25);
    -- elem:setMaxLength(6);
    -- show the window natively
    window:Show("my_window", nil, "_lt", 0,0, 200, 200);
end
```

Creating Custom MCML v2 Controls

This is our custom control `MyApp.Controls.MyCustomControl`, whose implementation is defined in `test_pe_custom.lua`

```
<pe:mcml>
<script type="text/npl" refresh="false">
<![CDATA[
]]>
</script>
    <pe:custom src="test_pe_custom.lua" classns="MyApp.Controls.MyCustomControl"␣
↪style="width:200px;height:200px;">
    </pe:custom>
</pe:mcml>
```

Now in `test_pe_custom.lua`, we create its implementation.

```
local MyCustomControl = commonlib.inherit(commonlib.gettable("System.Windows.UIElement
↪"), commonlib.gettable("MyApp.Controls.MyCustomControl"));

function MyCustomControl:paintEvent(painter)
    painter:SetPen("#ff0000");
    painter:DrawText(10,10, "MyCustomControl");
end
```

Preview and Debug Layout

Sometimes, we want to change the code and preview the result without restarting the whole application. If your windows contains no external logics and uses mcml v1 exclusively, you can preview using MCML browser, simply press `Ctrl+F3`.

In most cases, you may need to write temporary or persistent test code for your new window class with data connected. This usually involves delete the old window object and create a new one like below. Run it repeatedly in NPL code wiki's console to preview your updated mcml code.

```
    -- remove old window
    local window = commonlib.gettable("test.window")
    if(window and window.CloseWindow) then
        window:CloseWindow(true);
    end

    -- create a new window
```

```
    NPL.load("(gl)script/ide/System/Windows/Window.lua");
    local Window = commonlib.gettable("System.Windows.Window")
    local window = Window:new();
    window:Show({
        url="script/ide/System/test/test_mcml_page.html",
        alignment="_lt", left = 0, top = 0, width = 200, height = 400,
    });
    -- keep a reference for refresh
    test.window = window;
```

CHAPTER 47

MCML Markup Language For 2D GUI

MCML or Micro Cosmos Markup Language is a meta language written in NPL since 2008, since then it has become the standard of writing 2D user interface in NPL. MCML is inspired by early version of ASP.net, however it uses local architecture for local user interface.

Quick Sample

Here is an example of mcml page in a `.html` file, such as `source/HelloWorldMCML/mcml_window.html`. The file extension only makes it possible for you to preview the page in other html code editor.

```
<pe:mcml>
<script refresh="false" type="text/npl" src="mcml_window.lua"><![CDATA[
    function OnClickOK()
        local content = Page:GetValue("content") or "";
        _guihelper.MessageBox(":"..content);
    end
]]></script>
<div style="color:#33ff33;background-color:#808080">
    <div style="margin:10px;">
        Hello World from HTML page!
    </div>
    <div style="margin:10px;">
        <input type="text" name="content" style="width:200px;height:25px;" />
        <input type="button" name="ok" value="" onclick="OnClickOK"/>
    </div>
</div>
</pe:mcml>
```

To render the page, one can launch it with a window like below. See System.Window for details.

```
    NPL.load("(gl)script/ide/System/Windows/Window.lua");
    local Window = commonlib.gettable("System.Windows.Window")
    local window = Window:new();
    window:Show({
        url="source/HelloWorldMCML/mcml_window.html",
        alignment="_lt", left = 300, top = 100, width = 300, height = 400,
    });
```

MCML V1 vs V2

There two implementations of MCML: `v1` and `v2`. The example above is given by v2, which is the preferred implementation. However, v1 is mature and stable implementation, v2 is new and currently does not have as many build-in controls as v1. We are still working on v2 and hope it can be 100% compatible and powerful with v1. This article is only about v2. See last section for v1.

Difference between v1 and v2

- v1 uses NPL's build-in GUI objects written in C++, i.e. ParaUIObject like `button`,`container`,`editbox`. Their implementation is hidden from NPL scripts and user IO is exposed to NPL script via callback functions.
- v2 uses NPL script to draw all the GUI objects using 3d-triangle drawing API and handles user input in NPL script. Therefore the user has complete control over the look of their control in NPL script. One can read the source code of v2 in main package.

Mcml tags

There are a number of tags which you can use to create interactive content. Besides, one can also create your own custom tags with UI controls, see System.Window for details. All mcml build-in tags are in the `pe:` xml namespace, which stands for ParaEngine or pe.

Standard html tags are mapped as follows:

- The `div` tag is mapped to `pe:div` class.
- plain text is mapped to `pe:text`.
- `input type=button` is mapped to `pe:button`, etc.

Examples

There are many mcml examples where you browse in the paracraft package. Most of them are written in mcml v1, however the syntax should be the same for mcml v2. Click to search paracraft package for mcml

Following is just some quick examples:

```
<pe:mcml>
<script type="text/npl" refresh="false">
<![CDATA[
globalVar = "test global var";
function OnButtonClick()
    _guihelper.MessageBox("refresh page now?", function()
        Page:SetValue("myBtn", "refreshed");
        Page:Refresh(0);
    end);
end

repeat_data = { {a=1}, {a=2} };
function GetDS()
    return repeat_data;
end
]]>
</script>
    <div align="right" style="background-color:#ff0000;margin:10px;padding:10px;min-
↪width:400px;min-height:150px">
```

```
        <div align="right" style="padding:5px;background:url(Texture/Aries/Common/
↪ThemeKid/btn_thick_hl_32bits.png:7 7 7 7)">
            <div style="float:left;background-color:#000000;color:#ffd800;font-size:
↪20px;">
                hello world<font color="#ff0000">fontColor</font>
            </div>
            <div style="float:left;background-color:#0000ff;margin-left:10px;width:
↪20px;height:20px;">
            </div>
        </div>
        <div valign="bottom">
            <div style="float:left;background-color:#00ff00;width:60px;height:20px;">
            </div>
            <div style="float:left;background-color:#0000ff;margin-left:10px;width:
↪20px;height:20px;">
            </div>
        </div>
    </div>
    <div>
        hello world<font color="#ff0000">fontColor</font>
        <span color="#ff0000">fontColor</span>
    </div>
    <span color="#ff0000">
        <%=Eval('globalVar')%>
        <%=document.write('hello world')%>
    </span>
    <pe:container style="padding:5px;background-color:#ff0000">
        button:<input type="button" name="myBtn" value="RefreshPage" onclick=
↪"OnButtonClick" style=""/>
        editbox:<input type="text" name="myEditbox" value="" style="margin-left:2px;
↪width:64px;height:25px"/><br/>
        pe_if: <pe:if condition='<%=globalVar=="test global var"%>'>true</pe:if>
    </pe:container>
    <pe:repeat value="item in repeat_data" style="float:left">
        <div style="float:left;"><%=item.a%></div>
    </pe:repeat>
    <pe:repeat value="item in GetDS()" style="float:left">
        <div style="float:left;"><%=item.a%></div>
    </pe:repeat>
    <pe:repeat DataSource='<%=GetDS()%>' style="float:left">
        <div style="float:left;"><%=a%></div>
    </pe:repeat>
</pe:mcml>
```

MCML Layout

Please see System.Window first for basic alignment types. There are over 12 different alignment types, which supports automatic resizing of controls according to parent window size.

- mcml v1 have a `<pe:container>` control which supports all alignment types via `alignment` attribute.
- mcml v1 and v2's `<div>` supports a limited set of alignment type via `align` and `valign` attribute, where you can align object to left or right. Please note, due to mcml v1's one pass rendering algorithm, `right` alignment only works when parent width is fixed sized. mcml v2 does not have this limitation because it uses two pass algorithm. A workaround for mcml v1 is to use `<pe:container>` instead of `<div>` and use `alignment` for right alignment.

Relative positioning

Relative position is a way to position controls without occupying any space. It can be specified with `style="position:relative;"`. If all controls in a container is relatively positioned, they can all be accurately positioned at desired location relative to their parent.

However, relative position is NOT recommended, because a single change in child control size may result in changing all other child controls' position values and if parent control's size changes, the entire UI may be wrong. The recommended way of position is "float:left" or line based position. `<div>` defaults to line based position, in which it will automatically add `line break` at end. Specify "float:left" to make it float. Many other controls default to "float:left", such as `<input>`, `<span>`. Some UI designer in special product may still prefer relative position despite of this.

MCML vs HTML

MCML only supports a very limited subset of HTML. The most useful layout method in mcml is `<div>` tag.

NPL Code Behind and Embedding

In MCML page, there are three ways to embed NPL code.

- Use `<script src="helloworld.lua">`, this will load the npl file relative to the current page file.
- Embed NPL code section inside `<script>` anywhere in the page file
- Embed code in xml attribute in quotations like this `attrname='<%= %>'`. Please note it depends on whether the tag implementation support it. For each attribute you can not mix code with text. Unlike in NPL web server page, all code in MCML are evaluated at runtime rather than preprocessed. This is very important.

```
<script type="text/npl" src="somefile.lua" refresh="false">
<![CDATA[
globalVar = "test global var";
function OnButtonClick()
    _guihelper.MessageBox("refresh page now?", function()
        Page:SetValue("myBtn", "refreshed");
        Page:Refresh(0);
    end);
end
repeat_data = { {a=1}, {a=2} };
function GetDS()
    return repeat_data;
end
]]>
</script>
<div style='<%=format("width:%dpx", 100) %>'></div>
```

Page Variable Scoping

Global variables or functions defined in inline script are local to the page and not visible outside the page. So consider using a separate code behind file if one wants to expose values to external environment or simply your code is too long.

There is a predefined variable called `Page`, which is accessible to the page's sandbox's environment.

Two-Way Databinding

Many HTML developers have a `jquery` mindset, where they like to manipulate the DOM (Document Object Model or XML node) directly, which is **NOT** supported in mcml. Instead, one should have a `databinding mindset` like in `angularjs`. In MCML v1, `DOM != Controls`. DOM is only a static template for creating controls in a page. Controls can never modify static template unless you write explicit code, which is really rare.

Most mcml tags only support one-way data binding from static DOM template to controls only at the time when controls are created (i.e. Page Refresh, see below). To read value back from controls, one must either listen to `onchange` event of individual control or manually call `Page:GetValue(name)` for a collection of controls in a callback function, such as when user pressed `OK` button.

Two-way data-binding is useful but hard to implement in any system. Some HTML framework like `angularjs` does it by using a timer to periodically check watched data model and hook into every control's `onchange` event. One can simulate it manually in mcml. In mcml v2, DOM is almost equal to controls, hence it is relatively easy to implement two-way data binding in v2 than in v1.

> Automatic two-way data-binding is not natively supported in both v1 and v2. But we are planning to support it in the near future. The following example shows how to do manual two-way binding in mcml.

```
<pe:mcml>
<p>Two-way databinding Example in mcml:</p>
<script type="text/npl" refresh="false"><![CDATA[
-- nodes are generated according to this data source object, ds is global in page␣
↪scope so that it is shared between page refresh
ds = {
    block1 = nil,
    block2 = nil,
};

function asyncFillBlock1()
    PullData();
    -- use timer to simulate async API, such as fetching from a remote server.
    commonlib.TimerManager.SetTimeout(function()
        -- simulate some random data from remote server
        ds.block1 = {};
        for i=1, math.random(2,8) do
            ds.block1[i] = {key="firstdata"..i, value=false};
        end
        ds.block2 = nil;
        ApplyData();
    end, 1000)
end

function asyncFillBlock2()
    PullData();
    commonlib.TimerManager.SetTimeout(function()
        ds.block2 = {};

        -- count is based on the number of checked item in data1(block1)
        local data1_count = 0;
        for i, data in pairs(ds.block1) do
            data1_count = data1_count + (data.value and 1 or 0);
        end

        -- generate block2 according to current checked item in data1(block1)
        for i=1, data1_count do
            ds.block2[i] = {key="seconddata"..i, value=false};
        end
```

```
        ApplyData();
    end, 1000)
end

-- apply data source to dom
function ApplyData()
    Page("#block1"):ClearAllChildren();
    Page("#block2"):ClearAllChildren();

    if(ds.block1) then
        local parentNode = Page("#block1");
        for i, data in pairs(ds.block1) do
            local node = Page("<span />", {attr={style="margin:5px"}, data.key });
            parentNode:AddChild(node);
            local node = Page("<input />", {attr={type="checkbox", name=data.key,␣
↪checked = data.value and "true"} });
            parentNode:AddChild(node);
        end
        local node = Page("<input />", {attr={type="button", value="FillBlock2",␣
↪onclick = "asyncFillBlock2"} });
        parentNode:AddChild(node);
    end

    if(ds.block2) then
        local parentNode = Page("#block2");
        for i, data in pairs(ds.block2) do
            local node = Page("<span />", {attr={style="margin:5px"}, data.key });
            parentNode:AddChild(node);
            local node = Page("<input />",  {attr={type="text", EmptyText="enter text
↪", name=data.key, value=data.value, style="width:80px;"} });
            parentNode:AddChild(node);
        end
    end

    Page:Refresh(0.01);
end

-- pull data from DOM
function PullData()
    if(ds.block1) then
        for i, data in pairs(ds.block1) do
            data.value = Page:GetValue(data.key);
        end
    end
    if(ds.block2) then
        for i, data in pairs(ds.block2) do
            data.value = Page("#"..data.key):value(); -- just another way of Page:
↪GetValue()
        end
    end
end

function OnClickSubmit()
    -- TODO: one can also use a timer and hook to every onchange event of controls to␣
↪automatically invoke Apply and Pull Data.
    -- Here we just need to manually call it.
    PullData();
    _guihelper.MessageBox(ds);
```

```
end
]]></script>
<div>
    <input type="button" value="FillBlock1" onclick="asyncFillBlock1()"/><br />
    block1: <div name="block1"></div>
    block2: <div name="block2"></div>
</div>
<input type="button" value="Submit" onclick="OnClickSubmit()" /><br />
</pe:mcml>
```

In above example, When user clicks `FillBlock1` button, some random data is filled in `ds.block1`, each data is displayed as checkbox, and the user can check any number of them and click the `FillBlock2` button. This time we will fill `ds.block2` with same number of checked items in `ds.block1`, and each item is displayed as a text input box. When user clicks `Submit` button, the current `ds` is displayed in a message box with user filled data.

We call `ApplyData` to rebuild all related DOM whenever `ds` is modified externally, and we call `PullData` in the first line of each callback function so that our `ds` data is always up to date with mcml controls. The above example is called manual two-way binding because we have to call `ApplyData` and `PullData` manually. One can also use a timer and hook to every onchange event of controls to automatically invoke `ApplyData` and `PullData`, so that we do not have to write them everywhere in our callback functions.

> There is an old two-way data binding code called BindingContext, which is only used in `<form>` tag in mcml, it may cause some strange behaviors when you are manipulating DOM.

Page Refresh

For dynamic pages, it is very common to change some variable in NPL script and refresh the page to reflect the changes.

`<script refresh='false'>` means that the code section will not be reevaluated when page is refreshed. Also note that global variables or functions are shared between multiple script sections or multiple page refreshes.

Please also note that code changes in inline script blocks are usually reparsed when window is rebuild. However, if one use code behind model, npl file is cached and you need to restart your application to take effect. It is good practise to write in inline script first, and when code is stable move them to a code behind page. This will also make your page load faster during frequent page refresh or rebuild.

Writing Dynamic Pages

MCML has its own way of writing dynamic pages. In short, mcml uses frequent full `page refresh` to build responsive UI layout. Please note that each mcml page refresh does NOT rebuild (reparse) the entire DOM tree again, it simply reuses the old DOM tree and re-execute any embedded code on them each time. If the code is marked as `refresh="false"` (see last section), its last calculated value will be used.

Generally, MCML v1 uses a single pass to render the entire web page. MCML v2 uses two passes (click here to see details).

Unlike HTML/Javascript(especially jquery) where programmers are encouraged to manipulate the document tree directly, in mcml, you are NOT advised to change the DOM with your own code (although we do provide a jquery like interface to its DOM). Instead, we encourage you to data-bind your code and create different code paths using conditional component like `<pe:if>`, when everything is set you do a full page refresh with `Page:refresh()`. This is a little bit like `AngularJS`, but we do it with real-time executed inline code. The difference is that `AngularJS` will compile all tags and convert everything from angular DOM to HTML DOM, but mcml just does one pass rendering of the mcml tags and execute the code as it converts DOM to real NPL controls.

Another thing is that unlike in AngularJS where data changes are watched and page is automatically refreshed, in mcml, one has to manually call `Page:refresh(delayTimeSeconds)` to refresh the whole page, such as when user pressed a button to change the layout. Please note, changing the DOM does not automatically refresh the page, one still need to call the refresh function. Mcml code logics is much simpler to understand and learn, while angularJS, although powerful, but with too many hidden dark-magic that programmers will never figure out on their own.

Finally, mcml page is NOT preprocessed into a second DOM tree before rendering like what NPL web server page or PHP does. Every tag in mcml is a real mcml control or PageElement and the DOM tree is persistent across page refresh. MCML uses a single pass to render all of them. This also makes full page `Page:refresh()` much faster than other technologies. Please note, mcml v2 has even faster page refresh than mcml v1, this is because in mcml v2, NPL controls are pure script objects, while in v1 they are ParaUIObject in ParaEngine. The former is much faster to create/destroy or even shared during page refresh depending on each PageElement's implementation.

MCML Controls

Remember mcml is meant to be replacement for tedious control creation in the whole application, not a stateless and isolated web page. Each mcml page is an integral part of the application, they run in the same thread and have access to everything in that thread. They usually have direct interactions (or data-binding) with the 3D scene and other UI controls. To get the underlying control, one can use `Page:FindControl(name)`

MCML Page:Refresh(DelayTime)

`Page:Refresh(DelayTime)` refresh the entire page after DelayTime seconds. Please note that during the delay time period, if there is another call to this function with a longer delay time, the actual refresh page activation will be further delayed Note: This function is usually used with a design pattern when the MCML page contains asynchronous content such as pe:name, etc. whenever an asynchronous tag is created, it will first check if data for display is available at that moment. if yes, it will just display it as static content; if not, it will retrieve the data with a callback function. In the callback function, it calls this Refresh method of the associated page with a delay time. Hence, when the last delay time is reached, the page is rebuilt and the dynamic content will be accessible by then.

- @param DelayTime: if nil, it will default to self.DefaultRefreshDelayTime (usually 1.5 second).
- tip: If one set this to a negative value, it may causes an immediate page refresh.

Page Styling and Themes

MCML has limited support of inline css in `style` or `class` parameter. MCML v1 and v2 support style tag, in which we can define inline or file based css in the form of a standard NPL table. Please note, the syntax is not `text/css`, but `text/mcss`. It is simply a table object with key name, value pairs. The key name can be tag class name or class name, compound names with `tag,id,class` filtering is not supported.

Here is an example of defining styles with different class names, all mcml tags can have zero or more class names.

```
<pe:mcml>
<style type="text/mcss" src="script/ide/System/test/test_file_style.mcss">
{
    color_red = { color = "#ff0000" },
    color_blue = { color = "#0000ff", ["margin-left"] = 10 },
    bold = { ["font-weight"]="bold"    }
}
</style>
<span class="color_blue bold">css class</span>
</pe:mcml>
```

External css files are loaded and cached throughout the lifetime of the application process. Its content is also NPL table, such as `script/ide/System/test/test_file_style.mcss`:

```
-- Testing mcml css: script/ide/System/test/test_file_style.mcss
{
["default"] = { color="#ffffff" },
["mobile_button"] = {
    background = "Texture/Aries/Creator/Mobile/blocks_UI_32bits.png;1 1 34 34:12 12␣
↪12 12",
},
color_red = { color = "#ff0000" },
color_blue = { color = "#0000ff", ["margin-left"] = 10, ["font-size"] = 16 },
bold = { ["font-weight"] = "bold"    },
["pe:button"] = {padding = 10, color="#00ff00"},
}
```

In most cases, we can style the object directly in style attributes using 9 tiled image file. See next section.

To change the style of unthemed tags or buildin classes, one must progammatically modify the following table:

- In mcml v1, there is a global table like this one
- In mcml v2, there is a class called StyleDefault

9-Tile Image

MCML supports a feature called image tiling. It can use a specified section of an image and divide the subsection into 9 images and automatically stretch the 5 middle images as the target resizes (4 images at the corner are unstretched). This is very useful to create good looking buttons with various sizes.

The syntax is like this: `<div style="background:url(someimage_32bits.png#0 0 64 21: 10 10 10 10);" >`

- `#left top width height` is optional, which specifies a sub region. if ignored it means the entire image.
- `:left top to_right to_bottom` means distance to left, top, right and bottom of the above sub region.
- `_32bits.png` means that we should use 32bits color, otherwise we will compress the color to save video memory
- Image size must of order of 2, like 2,4,8,16,64, 128, 256, etc. Because `directX/openGL` only supports order of 2 images. If your image file is not order of 2, they will be stretched and become blurred when rendered.

Please note, in css file or `<style>` tag, `#` can also be `;` when specifying a sub region.

Setting Key Focus

For mcml v1, one can set key focus by following code. It only works for text box.

```
local ctl = Page:FindControl(name);
if(ctl) then
   ctl:Focus();
end
```

For input text box, one can also use.

```
<input type="text" autofocus="true"  ... />
```

But there can only be one such control in a window. If there are multiple ones, the last one get focus. For example

```
<pe:mcml>
    <div>
        Text: <input type="text" name="myAutoFocusEditBox" />
    </div>
    ... many other div here...
    <!-- This should be the last in the page -->
    <script type="text/npl" refresh="true">
        local ctl = Page("#myAutoFocusEditBox"):GetControl()
        ctl:Focus();
        ctl:SetCaretPosition(-1);
    </script>
</pe:mcml>
```

Conclusion

MCML v2 implementation is not very complete yet. You are welcome to contribute your own tags or controls to our main package.

FAQ

Why large font size appears to be blurred in mcml?

MCML uses 3D API to do all the graphical drawing. For each font size, an image must be loaded into video memory. Thus by default, mcml may scale an existing font to prevent creating unnecessary image. To prevent this behavior, one can use `style="base-font-size:30;font-size:30"` this will always create a base font image for your current font setting. In general, an application should limit the number of font settings used in your graphical application.

References

Documentation For MCML V1

- mcml v1 Source Code, such as:
  - pe_html
  - pe_editor
  - pe_html_input
  - pe_gridview: more examples
  - pe_treeview
  - pe_design
  - pe_component
  - pe_script
- Test cases(Examples) for mcml v1

More over, many existing GUI of paracraft are actually created with mcml v1. Search the source code for any HTML pages for examples.

CHAPTER 48

3D Programming

NPL/ParaEngine provides rich set of graphical API and libraries to create sophisticated interactive 3D application. ParaEngine used to be a full-featured computer game engine, it is now built-in with NPLRuntime. ParaEngine implementation is C/C++ based. All of its API can be accessed via NPL scripts.

Example 3D Projects

- Paracraft: is a 3D animation software written completely in NPL with over half million lines of NPL code.
- Haqi: is a 3D MMORPG written completely in NPL with over 1 million lines of NPL code.

CHAPTER 49

File Format

ParaEngine/NPL support several build-in file format. Some is used for 3d models, some for 3d world

3D Model file format

ParaEngine supports following built-in or standard file format.

- ParaX format (*.x): This is our build-in file format for advanced animated characters, particles, etc. One needs to download and install ParaEngine exporter plugin for 3dsmax 9 (32bits/64bits). However, we are not supporting the latest version of 3dsmax. Please consider using FBX file format.
- FBX format (`*.fbx`): FBX is the universal file format for AutoDesk product like MAYA/3dsmax. It is one of the most supported lossless file format in the industry. Click here for how to use FPX. (Please note model size and embedded texture matters)

  we only support FBX version 2013 or above. Version 2014, 2015 are tested.
- BMAX (*.bmax): This is our built-in block max file format, which is used exclusively by Paracraft application for storing static and animated blocks.
- STL format (*.stl): This is a file format used for 3d printing.

BMax file format

BMAX is short for Block Max, it is a file format used exclusively in Paracraft for storing and exchanging block data. In paracraft, the world is made up of blocks, each block may contain `x,y,z,block_id,block_data,custom_data`

- `x,y,z` is block position
- `block_id` is type id of the block, see block_types.xml for a complete list of block ids.
- `block_data` is a 32bits data of the block, it usually denotes the orientation or type of the block.
- `custom_data` can be any NPL table object that stores additional data of the block. Most static blocks does not contain custom_data, however, blocks like movie block, command blocks will save animation data and commands in this place.

You can select some blocks in paracraft and save them to bmax file to examine a real bmax file, it should be self-explanatory. Following is an example bmax file(containing a button, 2 movie blocks, a repeater and a wire):

```
<pe:blocktemplate>
    <pe:blocks>{
{-2,0,-3,105,5,},

{-2,0,-2,228,[6]={{name="cmd","/t 6 /end",},{{"{timeseries={lookat_z={times={0,5348,},
↪data={20001.56125,20004.65625,},ranges={{1,2,},},type=\"Linear\",name=\"lookat_z\",}
↪,eye_liftup={times={0,5348,},data={0.28568,0.26068,},ranges={{1,2,},},type=\"Linear\
↪",name=\"eye_liftup\",},lookat_x={times={0,5348,},data={20000.51959,20000.58203,},
↪ranges={{1,2,},},type=\"Linear\",name=\"lookat_x\",},eye_rot_y={times={0,5348,},
↪data={-3.11606,-3.11668,},ranges={{1,2,},},type=\"LinearAngle\",name=\"eye_rot_y\",}
↪,is_fps={times={0,5348,},data={0,0,},ranges={{1,2,},},type=\"Discrete\",name=\"is_
↪fps\",},lookat_y={times={0,5348,},data={-127.08333,-127.08333,},ranges={{1,2,},},
↪type=\"Linear\",name=\"lookat_y\",},eye_dist={times={0,5348,},data={8,8,},ranges={
↪{1,2,},},type=\"Linear\",name=\"eye_dist\",},has_collision={times={0,5348,},data={1,
↪1,},ranges={{1,2,},},type=\"Discrete\",name=\"has_collision\",},eye_roll={times={0,
↪5348,},data={0,0,},ranges={{1,2,},},type=\"LinearAngle\",name=\"eye_roll\",},},}",
↪name="slot",attr={count=1,id=10061,},},{"{timeseries={time={times={},data={},ranges=
↪{},type=\"Linear\",name=\"time\",},music={times={},data={},ranges={},type=\
↪"Discrete\",name=\"music\",},tip={times={},data={},ranges={},type=\"Discrete\",
↪name=\"tip\",},movieblock={times={},data={},ranges={},type=\"Discrete\",name=\
↪"movieblock\",},cmd={times={},data={},ranges={},type=\"Discrete\",name=\"cmd\",},
↪blocks={times={},data={},ranges={},type=\"Discrete\",name=\"blocks\",},text={times=
↪{},data={},ranges={},type=\"Discrete\",name=\"text\",},},}",name="slot",attr=
↪{count=1,id=10063,},},name="inventory",{"{timeseries={blockinhand={times={},data={},
↪ranges={},type=\"Discrete\",name=\"blockinhand\",},x={times={0,},data={20000.51959,}
↪,ranges={{1,1,},},type=\"Linear\",name=\"x\",},pitch={times={},data={},ranges={},
↪type=\"LinearAngle\",name=\"pitch\",},y={times={0,},data={-127.08333,},ranges={{1,1,
↪},},type=\"Linear\",name=\"y\",},parent={times={},data={},ranges={},type=\
↪"LinearTable\",name=\"parent\",},roll={times={},data={},ranges={},type=\
↪"LinearAngle\",name=\"roll\",},block={times={},data={},ranges={},type=\"Discrete\",
↪name=\"block\",},scaling={times={},data={},ranges={},type=\"Linear\",name=\"scaling\
↪",},gravity={times={},data={},ranges={},type=\"Discrete\",name=\"gravity\",},
↪HeadUpdownAngle={times={},data={},ranges={},type=\"Linear\",name=\"HeadUpdownAngle\
↪",},anim={times={},data={},ranges={},type=\"Discrete\",name=\"anim\",},bones={R_
↪Forearm_rot={times={0,34,1836,},data={{0.00024,0.00175,-0.01261,0.99992,},{0.00024,
↪0.00175,-0.01261,0.99992,},{0.00022,-0.05946,0.42959,0.90107,},},ranges={{1,3,},},
↪type=\"Discrete\",name=\"R_Forearm_rot\",},R_UpperArm_rot={times={0,34,1836,},data={
↪{-0.01933,-0.00286,0.03036,0.99935,},{-0.01802,-0.03834,0.32796,0.94375,},{-0.0137,-
↪0.01077,0.14324,0.98954,},},ranges={{1,3,},},type=\"Discrete\",name=\"R_UpperArm_
↪rot\",},isContainer=true,},speedscale={times={},data={},ranges={},type=\"Discrete\",
↪name=\"speedscale\",},assetfile={times={0,},data={\"actor\",},ranges={{1,1,},},
↪type=\"Discrete\",name=\"assetfile\",},skin={times={},data={},ranges={},type=\
↪"Discrete\",name=\"skin\",},z={times={0,},data={20001.56125,},ranges={{1,1,},},
↪type=\"Linear\",name=\"z\",},facing={times={},data={},ranges={},type=\"LinearAngle\
↪",name=\"facing\",},HeadTurningAngle={times={},data={},ranges={},type=\"Linear\",
↪name=\"HeadTurningAngle\",},name={times={0,},data={\"actor3\",},ranges={{1,1,},},
↪type=\"Discrete\",name=\"name\",},opacity={times={},data={},ranges={},type=\"Linear\
↪",name=\"opacity\",},},tooltip=\"actor3\",}",name="slot",attr={count=1,id=10062,},},
↪},name="entity",attr={bz=19201,bx=19200,class="EntityMovieClip",item_id=228,by=5,},}
↪,},

{-2,0,-1,197,3,},
{-2,0,0,189,},
{-2,0,1,228,[6]={{name="cmd","/t 30 /end",},{name="inventory",{name="slot",attr=
↪{count=1,id=10061,},},},name="entity",attr={bz=19204,bx=19200,class="EntityMovieClip
↪",item_id=228,by=5,},},},}

    </pe:blocks>
```

```
</pe:blocktemplate>
```

For user manual of using movie blocks, please see the courses in

- http://www.paracraft.cn/learn/movieblockcourses?lang=zh

ParaX File format

ParaX is binary file format used exclusively in NPL Runtime for animated 3D character asset.ParaX is like a concise version of FBX file (FBX is like BMAX file used by autodesk 3dsmax/maya, ...) NPL Runtime also support reading FBX file and load as ParaXModel object in C++, for user guide see FBX

- BMAX to ParaX converter in C++ (Lacking support for animation data in movie block) https://github.com/LiXizhi/NPLRuntime/tree/master/Client/trunk/ParaEngineClient/BMaxModel
- ParaX models:
  - https://github.com/LiXizhi/NPLRuntime/blob/master/Client/trunk/ParaEngineClient/ParaXModel/XFileCharModelParser.h
  - https://github.com/LiXizhi/NPLRuntime/blob/master/Client/trunk/ParaEngineClient/3dengine/ParaXSerializer.h
  - https://github.com/LiXizhi/NPLRuntime/blob/master/Client/trunk/ParaEngineClient/3dengine/ParaXSerializer.h

BMAX movie blocks to ParaX File format Conversion Details

- locate all movie blocks in BMAX file and ignore all other blocks. Movie blocks needs to be in the same order of 3D space (as repeater and wires indicates)
- The first movie block is always animation id 0 (idle animation)
- The second movie block is always animation id 4 (walk animation)
- The animation id of the third or following movie blocks can be specified in its movie block command
- The BMAX model(s) in the first movie block is used for defining bones and mesh (vertices, etc), please see this code for how to translate blocks into bones, vertices and sub meshes. BMAX models in all other movie blocks are ignored.
- Animation data in movie blocks are used to generate bone animations for each animation id in the final ParaX-Model.

3D World File Format

ParaEngine/NPL can automatically save or load 3D scene object to or from disk files.

To create an empty world, one can use

```
local worldpath = "temp/clientworld";
ParaIO.DeleteFile(worldpath.."/");
ParaIO.CreateDirectory(worldpath.."/");
ParaWorld.NewEmptyWorld(worldpath, 533.3333, 64);
```

3D world is tiled. Each tile is usually 512*512 meters. For historical reasons, it is 533.3333 by default. You will notice three files, after calling above function.

`temp/clientworld/worldconfig.txt` which is the entry file for 3d world, its content is like

```
-- Auto generated by ParaEngine
type = lattice
TileSize = 533.333313
(0,0) = temp/clientworld/flat.txt
(0,1) = temp/clientworld/flat.txt
...
```

It contains a mapping from tile (x,y) to tile configuration file.

`temp/clientworld/flat.txt` is configuration file for a single terrain tile.

```
-- auto gen by ParaEngine
Heightmapfile = temp/clientworld/flat.raw
MainTextureFile = terrain/data/MainTexture.dds
CommonTextureFile = terrain/data/CommonTexture.dds
Size = 533.333313
ElevScale = 1.0
Swapvertical = 1
HighResRadius = 30
DetailThreshold = 50.000000
MaxBlockSize = 64
DetailTextureMatrixSize = 64
NumOfDetailTextures = 0
```

Global Terrain Format

Global Terrain Format is a LOD triangle tile based terrain engine. The logical tile size is defined by 3d world configuration file, such as 533.333, however, its real dimension is always 512*512 and saved as a `*.raw` file.

Block World File Format

ParaEngine has a build-in block engine for rendering 3D blocky world. Block world is also tiled. The logical tile size is defined by 3d world configuration file, such as 533.333, however, its real dimension is always `512x512` and saved as a binary block region file in `*.raw` extension. Please note that block world region file `*.raw` is different from global terrain's `*.raw` file. The latter is just height maps.

Currently, each block may contain `x,y,z,block_id,block_data[,custom_data]`, currently `custom_data` is not supported in block region `*.raw` file. Block region file is encoded (compressed) using `increase-by-integer` algorithm, which will save lots of disk space if blocks are of the same type in the tiled region.

See BlockRegion.cpp for details

References

- http://www.paracraft.cn/ : see bmax model tutorial video
- https://github.com/LiXizhi/NPLRuntime/wiki
- https://github.com/LiXizhi/STLExporter
- http://wikicraft.cn/wiki/mod/packages : see STLExporter

- BMAX to ParaX converter in C++ (Lacking support for animation data in movie block) https://github.com/LiXizhi/NPLRuntime/tree/master/Client/trunk/ParaEngineClient/BMaxModel
- ParaX models:
  - https://github.com/LiXizhi/NPLRuntime/blob/master/Client/trunk/ParaEngineClient/ParaXModel/XFileCharModelParser.h
  - https://github.com/LiXizhi/NPLRuntime/blob/master/Client/trunk/ParaEngineClient/3dengine/ParaXSerializer.h

CHAPTER 50

3D Scene

There are a number of 3D objects which one can create via scripting API, such as mesh, physics mesh, sky box, camera, biped character, light, particles, containers, overlays, height map terrain, blocks, animations, shaders, 3D assets, etc.

Triangle Limitations

One can set the maximum number of total triangles of animated characters in the 3d scene per frame, like below

```
ParaScene.GetAttributeObject():SetField("MaxCharTriangles", 150000);
```

Faraway objects exceeding this value will be skipped during rendering. Another way of reducing triangle count is via level-of-detail (LOD) when loading mesh or characters. Currently, one must manually provide all LOD of mesh when loading a 3d file.

Rendering Pipeline

Currently all fixed function, shaders and deferred shading share the same predefined rendering pipelines. The render order can be partially affected by `RenderImportance`, `RenderOrder`, several `Shader files` property. But in general, it is a fixed general purpose rendering pipeline suitable in most situations.

The source code of the pipeline is hard-coded in `SceneObject.cpp's AdvanceScene()` function.

Here is what it does:

For each active viewport, we do the following:

- traverse the scene and quad-tree tile manager and call `PrepareRender()` for each visible scene object, which will insert the object into a number of predefined global render queues.
- `PIPELINE_3D_SCENE`
  - draw `MiniSceneGraph` with local cameras and render into textures
  - draw owner-draw objects like render targets.
  - draw all m_mirrorSurfaces into local textures
  - tesselate `global terrain` according to current camera
  - render shadow map for shadow casters queue, and other global object like blocks that cast shadows.
  - draw block engine multiframe world texture

  - render `global terrain`
  - draw opaque blocks in block engine
  - draw alpha-test enabled blocks in block engine
  - draw static meshes in big mesh queue from front to back
  - draw static meshes in small mesh queue from back to front
  - draw sprite objects queue
  - draw animated characters queue
  - render selection queue
  - render current sky object
  - draw block engine multiframe world texture on sky
  - draw missile object queue
  - draw block engine water reflection pass
  - block engine deferred shading post processing for opaque objects
  - draw batched transparent particles
  - block engine draw all alpha blended blocks
  - draw transparent animated characters in transparent biped queue
  - block engine deferred shading post processing for alpha blended object
  - draw block engine deferred lights
  - draw all head-on display
  - draw simulated global ocean surface
  - draw transparent static mesh objects in transparent mesh queue
  - draw transparent triangle face groups
  - draw objects in `post render queue`
  - draw particle systems
  - invoke custom post rendering shaders in NPL script for 3d scene
  - draw water waves
  - draw helpers for physics world debugging
  - draw helpers for bounding boxes of scene objects
  - draw portal system
  - draw overlays
- `PIPELINE_UI`
  - render all visible GUI objects by traversing the `GUIRoot` object.
- `PIPELINE_POST_UI_3D_SCENE`
  - only `MiniSceneGraph` whose `GetRenderPipelineOrder() == PIPELINE_POST_UI_3D_SCENE` will be rendered in this pipeline
- `PIPELINE_COLOR_PICKING`

Pipeline customization

- `RenderImportance` property of ParaObject only affects render order in a given queue.
- `RenderOrder` property of ParaObject will affect which queue the object goes into, as well as the order inside the queue. However, only `RenderOrder>100` is used to insert objects to `post render queue`.

For example, the following code will ensure the two objects are rendered last and ztest is disabled.

```
    local asset = ParaAsset.LoadStaticMesh("","model/common/editor/z.x")
    local obj = ParaScene.CreateMeshPhysicsObject("blueprint_center", asset, 1,1,1,␣
↪false, "1,0,0,0,1,0,0,0,1,0,0,0");
    obj:SetPosition(ParaScene.GetPlayer():GetPosition());
    obj:SetField("progress",1);
    obj:GetEffectParamBlock():SetBoolean("ztest", false);
    obj:SetField("RenderOrder", 101)
    ParaScene.Attach(obj);

    local player = ParaScene.CreateCharacter ("MyPlayer1", ParaAsset.LoadParaX("",
↪"character/v3/Elf/Female/ElfFemale.x"), "", true, 0.35, 0, 1.0);
    local x,y,z = ParaScene.GetPlayer():GetPosition()
    player:SetPosition(x+1,y,z);
    player:SetField("RenderOrder", 100)
    player:GetEffectParamBlock():SetBoolean("ztest", false);
    ParaScene.Attach(player);
```

CHAPTER 51

Camera

ParaEngine has build-in C++ support for a very versatile camera object, called AutoCamera. AutoCamera is the default camera used when you are inside a 3d world.

It can be configured in various ways to handle most key/mouse input for you, as well as controlling the biped object that it is focusing on.

To get or set current camera's attributes, one can use following code. Use Object Browser in NPL Code Wiki to see all of its attributes. Camera is a child of Scene object.

```
local attr = ParaCamera.GetAttributeObject()
```

Control Everything In Your Own Code

In most cases, we can focus the camera to a Biped player object in the scene to control the motion of the player directly. Trust me, writing a robust and smooth player camera controller is difficult, at least 3000 lines of code. If you really want to handle everything yourself, there is an option. Create a dummy invisible object to let the camera to focus to. And then set `IsControlledExternally` property to true for the dummy object.

```
your_dummy_obj:SetField("IsControlledExternally", true)
```

This way, the auto camera will not try to control the focused dummy object any more. And it is up to you to control the player, such as using `WASD` or arrow keys for movement and mouse to rotate its facing, etc.

Code Examples

Following are from `ParaEngineExtension.lua`. They are provided as an example.

```
-- set the look at position of the camera. It uses an invisible avatar as the camera␣
↪look at position.
-- after calling this function, please call ParaCamera.SetEyePos(facing, height,␣
↪angle) to change the camera eye position.
function ParaCamera.SetLookAtPos(x, y, z)
    local player = ParaCamera.GetDummyObject();
    player:SetPosition(x, y - 0.35, z);
    player:ToCharacter():SetFocus();
end

function ParaCamera.GetLookAtPos()
```

```
    return unpack(ParaCamera.GetAttributeObject():GetField("Lookat position", {0,0,0}
↪));
end
-- it returns polar coordinate system.
-- @return camobjDist, LifeupAngle, CameraRotY
function ParaCamera.GetEyePos()
    local att = ParaCamera.GetAttributeObject();
    return att:GetField("CameraObjectDistance", 0), att:GetField("CameraLiftupAngle",␣
↪0), att:GetField("CameraRotY", 0);
end

-- create/get the dummy camera object for the camera look position.
function ParaCamera.GetDummyObject()
    local player = ParaScene.GetObject("invisible camera");
    if(player:IsValid() == false) then
        player = ParaScene.CreateCharacter ("invisible camera", "", "", true, 0, 0,␣
↪0);
        --player:GetAttributeObject():SetField("SentientField", 0);--senses nobody
        player:GetAttributeObject():SetField("SentientField", 65535);--senses␣
↪everybody
        --player:SetAlwaysSentient(true);--senses everybody
        player:SetDensity(0); -- make it flow in the air
        player:SetPhysicsHeight(0);
        player:SetPhysicsRadius(0);
        player:SetField("SkipRender", true);
        player:SetField("SkipPicking", true);
        ParaScene.Attach(player);
        player:SetPosition(0, 0, 0);
    end
    return player;
end

-- set the camera eye position by camera object distance, life up angle and rotation␣
↪around the y axis. One must call ParaCamera.SetLookAtPos() before calling this␣
↪function.
-- e.g.ParaCamera.SetEyePos(5, 1.3, 0.4);
function ParaCamera.SetEyePos(camobjDist, LifeupAngle, CameraRotY)
    local att = ParaCamera.GetAttributeObject();
    att:SetField("CameraObjectDistance", camobjDist);
    att:SetField("CameraLiftupAngle", LifeupAngle);
    att:SetField("CameraRotY", CameraRotY);
end
```

CHAPTER 52

Block Engine

The block engines can manage multiple `BlockWorld` object.

BlockWorld

Each `BlockWorld` represents a block world with at most `32000x32000x256`, where 256 is height of the world. Each BlockWorld will dynamically and asynchronously load `BlockRegion` on demand.

BlockRegion

It manages 512x512x256 blocks, which are saved into a single file.

BlockChunk

It caches model and light Data for `16x16x16` region. Each chunk is converted and added to a queue into `BlockRenderTask` for sorting and rendering.

BlockLightGrid

It calculates sun and block lighting in a separate thread and save the result into `BlockChunk` for rendering.

BlockModel

BlockModel is usually cube 3D model, but it is not a 3D object directly used in rendering, instead it is actually used in `BlockTemplate` to provide rendering and physics data.

CHAPTER 53

NPL Web Server

Click here for step-by-step tutorial to build a NPL web site.

NPL Runtime provides a build-in self-contained framework for writing web server applications in a way similar to PHP, yet with asynchronous API like NodeJs.

Why Write Web Servers in NPL?

Turning Async Code Into Synchronous Ones

Many tasks on the server side has asynchronous API like database operations, remote procedure call, background tasks, timers, etc. Asynchronous API frees the processing thread from io-bound or long running task. Using asynchronous API (like those in `NodeJs`) makes web server fast, but difficult to write, as there are too many function callbacks that breaks the otherwise sequential and flat programming code.

In NPL page file, any asynchronous API can be used as synchronous ones via `resume/yield`, so that constructing a web page is like writing a document sequentially like in `PHP`. Under the hood, it is still asynchronous API and the same worker thread can process thousands of requests per second even some of them takes a long time to finish. Following is a quick example show the idea.

```
<?
local value = "Not Set"
-- any async function with callback
local mytimer = commonlib.Timer:new({callbackFunc = function(timer)
   value = "Set";
   resume();
end})
-- start the timer after 1000 milliseconds
mytimer:Change(1000, nil)

-- async wait when job is done
yield();
assert(value == "Set");
```

Please note, `resume/yield` is NOT the ones as in multithreaded programming where locks and signals are involved. It is a special `coroutine` internally; the code is completely lock-free and the thread is `NOT` waiting but processing other URL requests. Please see NPLServerPage for details.

Self-Contained Client/Server Solution

NPL Web Server is fast and very easy to deploy on all platforms (no external dependencies even for databases). You may have a very sophisticated 3d client application, which may also be servers and/or web servers, such as Paracraft.

We believe, every software either running on central server farm or endpoints like home or mobile devices should be able to provide services via TCP connection and/or HTTP (web server). In many of our applications, a single software acts both as client and server. Very few existing programming language provides an easy way or architecture for sharing client and server side code or variables in a lock-free fashion due to multi-threaded (multi-process) nature of those web frameworks. Moreover, a server application is usually hard to deploy and config, making it difficult to install on home computers or mobile devices.

NPL is a language to solve these problems and provides rich client side API as well as a web server framework that can service as many requests as professional web servers while sharing the entire runtime environment for your client and server code. See below.

Programming Model of NPL web server

NPL Runtime provides a build-in framework for writing web server applications in a way similar to PHP.

> NPL Web Server recommends the async programming model like Nodejs, but without losing the simplicity of synchronous mixed code programming like in PHP.

Following is an example `helloworld.page` server page.

| |query database and wait for database result | MVC Render | |—-|———————————————————-|———| | duration |95% | 5% |

The processing of a web page usually consists of two phases.

- One is fetching data from database engine, which usually takes over 95% of the total time.
- The other is page rendering, which is CPU-bound and takes only 5% of total request time.

With NPL's `yield` method, it allows other web requests to be processed concurrently in the 90% interval while waiting database result on the `same` system-level thread. See following code to see how easy to mix async-code with template-based page rendering code. This allows us to serve `5000 requests/sec` in a single NPL thread concurrently, even if each request takes 30ms seconds to fetch from database.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
↪xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title>NPL server page test. </title>
</head>
<body>
<p>
    <?npl
        echo "Hi, I'm a NPL script!";
    ?>
</p>

<?
-- connect to TableDatabase (a NoSQL db engine written in NPL)
db = TableDatabase:new():connect("database/npl/", function() end);

-- insert 5 records to database asynchronously.
local finishedCount = 0;
```

```
for i=1, 5 do
    db.TestUser:insertOne({name=("user"..i), password="1"}, function(err, data)
        finishedCount = finishedCount + 1;
        if(finishedCount == 5) then
            resume();
        end
    end);
end
yield(); -- async wait when job is done

-- fetch all users from database asynchronously.
db.TestUser:find({}, function(err, users)  resume(err, users); end);
err, users = yield(); -- async wait when job is done
?>

<?npl for i, user in ipairs(users) do ?>
    i = <?=i?>, name=<? echo(user.name) ?> <br/>
<?npl end ?>

<p>
    1.  <?npl echo 'if you want to serve NPL code in XHTML or XML documents, use␣
↪these tags'; ?>
</p>
<p>
    2.  <? echo 'this code is within short tags'; ?>
    Code within these tags <?= 'some text' ?> is a shortcut for this code <? echo
↪'some text' ?>
</p>
<p>
    3.  <% echo 'You may optionally use ASP-style tags'; %>
</p>

<% nplinfo(); %>

<% print("<p>filename: %s, dirname: %s</p>", __FILE__, dirname(__FILE__)); %>

<%
some_global_var = "this is global variable";

-- include file in same directory
include("test_include.page");
-- include file relative to web root directory. however this will not print, because␣
↪we use include_once, and the file is already included before.
include_once("/test_include.page");
-- include file using disk file path
local result = include(dirname(__FILE__).."test_include.page");

-- we can also get the result from included file
echo(result);
%>

</body>
</html>

<%
--assert(false, "uncomment to test runtime error");

function test_exit()
```

```
    exit();
end
test_exit();
assert(false, "can not reach here");
%>
```

Content of `test_include.page` referenced in above server page is:

```
<?npl

echo("<p>this is from included file: "..(some_global_var or "").."</p>");

-- can also has return value.
return "<p>from test_include.page</p>";
```

The output of above server page, when viewed in a web browser, such as `http://localhost:8099/helloworld` is

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
↪xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title>NPL server page test. </title>
</head>
<body>
<p>
    Hi, I'm a NPL script!</p>

    i = 1, name=user1 <br/>
    i = 2, name=user2 <br/>
    i = 3, name=user3 <br/>
    i = 4, name=user4 <br/>
    i = 5, name=user5 <br/>
<p>
    1.  if you want to serve NPL code in XHTML or XML documents, use these tags</p>
<p>
    2.  this code is within short tags    Code within these tags some text is a␣
↪shortcut for this code some text</p>
<p>
    3.  You may optionally use ASP-style tags</p>

<p>NPL web server v1.0</p><p>site url: http://localhost:8099/</p><p>your ip: 127.0.0.1
↪</p><p>{
<br/>["Host"]="localhost:8099",
<br/>["rcode"]=0,
<br/>["Connection"]="keep-alive",
<br/>["Accept"]="text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;
↪q=0.8",
<br/>["Accept-Encoding"]="gzip, deflate, sdch",
<br/>["method"]="GET",
<br/>["body"]="",
<br/>["tid"]="~1",
<br/>["url"]="/helloworld.page",
<br/>["User-Agent"]="Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML,␣
↪like Gecko) Chrome/48.0.2564.116 Safari/537.36",
<br/>["Upgrade-Insecure-Requests"]="1",
```

```
<br/>["Accept-Language"]="en-US,en;q=0.8",
<br/>}
<br/></p><p>filename: script/apps/WebServer/test/helloworld.page, dirname: script/
↪apps/WebServer/test/</p><p>this is from included file: this is global variable</p>
↪<p>this is from included file: this is global variable</p><p>from test_include.page
↪</p></body>
</html>
```

Web Server Source Code

- script/apps/WebServer: implementation of a web server (like Apache) in NPL.
- script/apps/WebServer/WebServer.lua: entry file
- script/apps/WebServer/admin: A php-like web site framework in NPL. See AdminSiteFramework for details

Introduction

The NPL web server implementation uses NPL's own messaging system, and is written in pure NPL scripts without any external dependencies. It allows you to create web site built with NPL Runtime on the back-end and `JavaScript/HTML5` on the client.

`NPL language service` plugins for visual studio (community edition is free) provides a good coding environment. See below:

Website Examples

- WebServerExample: a sample web site with NPL code wiki
- Wikicraft: full website example
- `script/apps/WebServer/admin` is a NPL based web site framework similar to the famous `WordPress.org`. It is recommended that you xcopy all files in it to your own web root directory and work from there. See AdminSiteFramework for details.
- `script/apps/WebServer/test` is a test site, where you can see the test code.

starting a web server programmatically

Starting server from a web root directory:

```
    NPL.load("(gl)script/apps/WebServer/WebServer.lua");
    WebServer:Start("script/apps/WebServer/test", "0.0.0.0", 8099);
```

Open `http://localhost:8099/helloworld.lua` in your web browser to test. See `log.txt` for details.

There can be a `webserver.config.xml` file in the web root directory. see below. if no config file is found, `default.webserver.config.xml` is used, which will redirect all request without extension to `index.page` and serve *.lua, *.page, and all static files in the root directory and it will also host NPL code wiki at http://127.0.0.1:8099.

starting a web server from command line

You can bootstrap a webserver using the buildin file, run following commmand:

```
npl bootstrapper="script/apps/WebServer/WebServer.lua"  port="8099"
```

- config: can be omitted which defaults to `config/WebServer.config.xml` in current directory

Web Server configuration file

More info, see `script/apps/WebServer/test/webserver.config.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<!-- web server configuration file: this node can be child node, thus embedded in␣
↪shared xml -->
<WebServer>
  <!--which HTTP ip and port this server listens to. -->
  <servers>
    <!-- @param host, port: which ip port to listen to. if * it means all. -->
    <server host="*" port="8099" host_state_name="">
      <defaultHost rules_id="simple_rule"></defaultHost>
      <virtualhosts>
        <!-- force "http://127.0.0.1/" to match to iternal npl_code_wiki site for␣
↪debugging  -->
        <host name="127.0.0.1:8099" rules_id="npl_code_wiki" allow='{"127.0.0.1"}'></
↪host>
      </virtualhosts>
    </server>
  </servers>
  <!--rules used when starting a web server. Multiple rules with different id can be␣
↪defined. -->
  <rules id="simple_rule">
    <!--URI remapping example-->
    <rule match='{"^[^%.]+$", "robots.txt"}' with="WebServer.redirecthandler" params='
↪{"/index.page"}'></rule>
    <!--npl script example-->
    <!--<rule match="%.lua$" with="WebServer.makeGenericHandler" params='{docroot=
↪"script/apps/WebServer/test", params={}, extra_vars=nil}'></rule>-->
    <rule match='{"%.lua$", "%.npl$"}' with="WebServer.npl_script_handler" params='%CD
↪%'></rule>
    <!--npl server page example-->
    <rule match="%.page$" with="WebServer.npl_page_handler" params='%CD%'></rule>
    <!--filehandler example, base dir is where the root file directory is. %CD% means␣
↪current file's directory-->
    <rule match="." with="WebServer.filehandler" params='{baseDir = "%CD%"}'></rule>
  </rules>
</WebServer>
```

Each server can have a default host and multiple virtual hosts. Each host must be associated with a rule_id. The rule_id is looked up in the rules node, which contains multiple rules specifying how request url is mapped to their handlers. `npl_code_wiki` is an internal rule_id which is usually used for host `http://127.0.0.1:8099/` for debugging purposes only, see above example code. If you changed your port number, you need to update the config file accordingly, otherwise `npl_code_wiki` will not be available.

Each rule contains three attributes: match, with and params.

- "match" is a regular expresion string or a array table of reg strings like shown in above sample code.

- "with" is a handler function which will be called when url matches "match". More precisely, it is handler function maker, that returns the real handler function(request, response) end. When rules are compiled at startup time, these maker functions are called just once to generate the actual handler function.
- "params" is an optional string or table that will be passed to the handler maker function to generate the real handler function.

The rules are applied in sequential order as they appeared in the config file. So they are usually defined in the order of redirectors, dynamic scripts, file handlers.

Handler Makers

Some common handlers are implemented in `common_handlers.lua`. Some of the most important ones are listed below

WebServer.redirecthandler

It is the url redirect handler. it generate a handler that replaces "match" with "params"

```
    <rule match="^[^%./]*/$" with="WebServer.redirecthandler" params='{"readme.txt"}'>
↪</rule>
```

The above will redirect `http://localhost:8080/` to `http://localhost:8080/readme.txt`.

WebServer.filehandler

It generate a handler that serves files in the directory specified in "params" or "params.baseDir".

```
    <rule match="." with="WebServer.filehandler" params='{baseDir = "script/apps/
↪WebServer"}'></rule>
    alternatively:
    <rule match="." with="WebServer.filehandler" params='script/apps/WebServer'></
↪rule>
```

The above will map `http://localhost:8080/readme.txt` to the disk file `script/apps/WebServer/readme.txt`

WebServer.npl_script_handler

Currently it is the same as WebServer.makeGenericHandler. In future we will support remote handlers that runs in another thread asynchrounously. So it is recommended for user to use this function instead of WebServer.makeGenericHandler serving request using npl files to generate dynamic response. This is very useful for REST-like web service in XML or json format.

```
    <rule match="%.lua$" with="WebServer.npl_script_handler" params='script/apps/
↪WebServer/test'></rule>
    alternatively:
    <rule match="%.lua$" with="WebServer.npl_script_handler" params='{docroot="script/
↪apps/WebServer/test"}'></rule>
```

The above will map `http://localhost:8080/helloworld.lua` to the script file `script/apps/WebServer/test/helloworld.lua`

Caching on NPL Web Server

1. We use `File Monitor API` to monitor all file changes in web root directory.
2. For dynamic page files: recompile dynamic page if changed. All compiled *.page code is always in cache. So multiple requests calling the same page file only compile once.
3. For static files: everything is cached in either original or gzip-format ready for direct HTTP response. All static files are served using a separate NPL thread to prevent IO affecting the main thread.

   for more advanced file caching, one may consider using `nginx` as the gateway load balancer.

Caching in Application Code

For caching in local thread, there is a helper class called mem_cache.

```
NPL.load("(gl)script/apps/WebServer/mem_cache.lua");
local mem_cache = commonlib.gettable("WebServer.mem_cache");
local obj_cache = mem_cache:GetInstance();
obj_cache:add("name", "value")
obj_cache:replace("name", "value1")
assert(obj_cache:get("name") == "value1");
assert(obj_cache:get("name", "group1") == nil);
obj_cache:add("name", "value", "group1")
assert(obj_cache:get("name", "group1") == "value");
```

Caching in different computer

TODO: `mem_cache` can be configured to read/write data from a remote computer.

CHAPTER 54

NPL Server Page

NPL server page is a mixed HTML/NPL file, usually with the extension `.page`.

How Does It Work?

At runtime time, server page is preprocessed into pure NPL script and then executed. For example

```
<html><body>
<?npl  for i=1,5 do ?>
   <p>hello</p>
<?npl  end ?>
</body></html>
```

Above server page will be pre-processed into following NPL page script, and cached for subsequent requests.

```
echo ("<html><body>");
for i=1,5 do
 echo("<p>hello</p>")
end
echo ("</body></html>");
```

When running above page script, `echo` command will generate the final HTML response text to be sent back to client.

Sandbox Environment

When a HTTP request come and redirected to NPL page handler, a special sandbox environment table is created, all page scripts related to that request is executed in this newly created sandbox environment. So you can safely create global variables and expect them to be uninitialized for each page request.

However, the sandbox environment also have read/write access to the global per-thread NPL runtime environment, where all NPL classes are loaded.

`NPL.load` vs `include`

In a page file, one can call `NPL.load` to load a given NPL class, such as `mysql.lua`; or one can also use the page command `include` to load another page file into sandbox environment. The difference is that classes loaded by `NPL.load` will be loaded only once per thread; where `include` will be loaded for every HTTP request handled by its worker thread. Moreover, NPL web server will monitor file changes for all page files and recompile them when modified by a developer; for files with `NPL.load`, you need to restart your server, or use special code to reload it.

Mixing Async Code with Yield/Resume

The processing of a web page usually consists of two phases.

- One is fetching data from database engine, which usually takes over 95% of the total time.
- The other is page rendering, which is CPU-bound and takes only 5% of total request time.

| |query database and wait for database result | MVC Render | |—-|———————————————————-|———| | duration |95% | 5% |

With NPL's `yield` method, it allows other web requests to be processed concurrently in the 90% interval while waiting database result on the `same` system-level thread. See following code to see how easy to mix async-code with template-based page rendering code. This allows us to serve `5000 requests/sec` in a single NPL thread concurrently, even if each request takes 30ms seconds to fetch from database.

Following is excerpt from our `helloworld.page` example.

```
<?
-- connect to TableDatabase (a NoSQL db engine written in NPL)
db = TableDatabase:new():connect("database/npl/", function() end);

-- insert 5 records to database asynchronously.
local finishedCount = 0;
for i=1, 5 do
    db.TestUser:insertOne({name=("user"..i), password="1"}, function(err, data)
        finishedCount = finishedCount + 1;
        if(finishedCount == 5) then
            resume();
        end
    end);
end
yield(); -- async wait when job is done

-- fetch all users from database asynchronously.
db.TestUser:find({}, function(err, users)  resume(err, users); end);
err, users = yield(true); -- async wait when job is done
?>

<?npl for i, user in ipairs(users) do ?>
    i = <?=i?>, name=<? echo(user.name) ?> <br/>
<?npl end ?>
```

Code Explanation: When the first `yield()` is called, the execution of page rendering is paused. It will be resumed by the result of a previous async task. In our case, when all five users have been inserted to our database, we will call `resume()`, which will immediately resume page execution from last yield (paused) code position.

Then we started another async task to fetch all users in the database, and called yield immediately to wait for its results. This time we passed some parameter `resume(err,users)`, everything passed to resume will be returned from yield() function. So `err,users = yield(true)` will return the `users` table when it returns.

Please note, we recommend you pass a boolean `err` as first parameter to `resume`, since all of our async API follows the same rule. Also note that we passed true to the second `yield` function, which tells to page renderer to output error and stop execution immediately. If you want to handle the error yourself, please pass nothing to yield like `err,users = yield()`

Page Commands

The following commands can only be called from inside an NPL page file. They are shortcut to long commands.

```
Following objects and functions can be used inside page script:
    request:   current request object: headers and cookies
    response:   current response object: send headers or set cookies, etc.
    echo(text):   output html
    __FILE__: current filename
    page: the current page (parser) object
    _GLOBAL: the _G itself

following are exposed via meta class:
    include(filename, bReload):  inplace include another script
    include_once(filename):  include only once, mostly for defining functions
    print(...):  output html with formated string.
    nplinfo():   output npl information.
    exit(text), die():   end the request
    dirname(__FILE__):   get directory name
    site_config(): get the web site configuration table
    site_url(path, scheme):
    addheader(name, value):
    file_exists(filename):
    log(obj)
    sanitize(text)  escape xml '<' '>'
    json_encode(value, bUseEmptyArray)   to json string
    json_decode(str)  decode from json string
    xml_encode(value)    to xml string
    include_pagecode(code, filename):  inplace include page code.
    get_file_text(filename)
    util.GetUrl(url, function(err, msg, data) end):
    util.parse_str(query_string):
    err, msg = yield(bExitOnError)  pause execution until resume() is called.
    resume(err, msg)  in async callback, call this function to resume execution from␣
↪last yield() position.
```

see script/apps/WebServer/npl_page_env.lua for detailed documentation.

Request/Response Object

- request object: current request object: headers and cookies
- response object: current response object: send headers or set cookies, etc.

Memory Cache and Global Objects

NPL web server has a built-in simple local memory cache utility. One can use it to store objects that is shared by all requests on the main thread. In future it may be configured to run on a separate server like `memcached`.

See below:

```
NPL.load("(gl)script/apps/WebServer/mem_cache.lua");
local mem_cache = commonlib.gettable("WebServer.mem_cache");
local obj_cache = mem_cache:GetInstance();
```

```
obj_cache:add("name", "value")
obj_cache:replace("name", "value1")
assert(obj_cache:get("name") == "value1");
assert(obj_cache:get("name", "group1") == nil);
obj_cache:add("name", "value", "group1")
assert(obj_cache:get("name", "group1") == "value");
```

Alternatively, one can also create global objects that is shared by all requests in the NPL thread, by using `commonlib.gettable()` method. Table objects created with `commonlib.gettable()` is different from `gettable()` in page file. The latter will create table on the local page scope which lasts only during the lifetime of a given http request.

File Uploader

NPL web server supports `multipart/form-data` by which one can upload binary file to the server. It is recommended to use a separate server for file upload, because it is IO bound and consumes bandwidth when file is very large.

Click here for a complete example of file uploader.

Here is the client side code to upload file as `enctype="multipart/form-data"`

```
<form name="uploader" enctype="multipart/form-data" class="form-horizontal" method=
↪"post" action="/ajax/fileuploader?action=upload">
  <input name="fileToUpload" id="fileToUpload" type="file" class="form-control"/>
  <input name="submit" type="submit" class="btn btn-primary" value="Upload"/>
</form>
```

Here is an example NPL server page code, the binary contents of the file is in `request:getparams()["fileToUpload"].contents`. The maximum file upload size allowed can be configured by NPL runtime attribute. The default value is 100MB.

```
local fileToUpload = request:getparams()["fileToUpload"]
if(fileToUpload and request:getparams()["submit"] and fileToUpload.name and␣
↪fileToUpload.contents) then
    local target_dir = "temp/uploads/" .. ParaGlobal.GetDateFormat("yyyyMMdd") .. "/";
    local target_file = target_dir .. fileToUpload.name;
    local fileType = fileToUpload.name:match("%.(%w+)$"); -- file extension

    -- check if file already exists
    if(file_exists(target_file)) then
        response:send({err = "Sorry, file already exists."});
        return
    end
    -- check file size
    if (fileToUpload.size and fileToUpload.size> 5000000) then
        response:send({err = "Sorry, your file is too large."});
        return
    end

    -- Allow certain file formats
    if(false and fileType ~= "jpg" and fileType ~= "png" and fileType~="txt" ) then
        response:send({err = "Sorry, only JPG, PNG & TXT files are allowed."});
        return
    end
```

```
    -- if everything is ok, try to save file to target directory
    ParaIO.CreateDirectory(target_dir);
    local file = ParaIO.open(commonlib.Encoding.Utf8ToDefault(target_file), "w");
    if(file:IsValid()) then
        file:write(fileToUpload.contents, #fileToUpload.contents);
        file:close();
        response:send({contents = target_file, name = fileToUpload.name, size =
↪fileToUpload.size, ["content-type"] = fileToUpload["content-type"], });
        return;
    else
        response:send({err = "can not create file on disk. file name invalid or disk
↪is full."});
    end
end
response:send({err = "unknown err"});
```

Server Side Redirection

See also RFC standard

```
<?npl
response:set_header("Location", "http://www.paracraft.cn/")
response:status(301):send_headers();
```

CHAPTER 55

NPL Admin Site Framework

`WebServer/admin` is a open source NPL-based web site framework. It comes with the `main` package and contains all source code of NPLCodeWiki. It is served as a demo and debugger for your own web application. It is also by default run side by side on `127.0.0.1:8099/` with your website on `localhost:8099`. See default.webserver.config.xml.

How to run NPL Admin Site

```
npl script/apps/WebServer/WebServer.lua
```

or you can run with more options like

```
npl bootstrapper="script/apps/WebServer/WebServer.lua"  port="8099"
```

Once started, you can visit the admin site from http://localhost:8099

Introduction

I have implemented it according to the famous `WordPress.org` blog site. The default theme template is based on "sensitive" which is a free theme of wordpress. "sensitive" theme has no dependency on media and can adjust display and layout according to screen width, which is suitable for showing on mobile phone device.

How to use

Usage 1: Use by reference (Recommended)

Basically, if you want to use admin framework without cloning the file, simply add following line to your rule file

```
    <!--wp framework related js, css, files-->
    <rule match="^/?wp%-" with="WebServer.filehandler" params='{baseDir = "script/
↪apps/WebServer/admin/"}'></rule>
```

In your page `index.page` file, you can include the main file of the wp framework, such as this:

```
<?npl

-- we will not load complete framework, but only ajax and helper functions
WP_USE_MINI_LOADER = true;
include_once("script/apps/WebServer/admin/wp-main.page");

-- call your router, such as
include_once("./myproj/routes.page");
```

Usage 2: Use by cloning

- copy everything in this folder to your own web site root, say /www/MySite/
- modify wp-content/database/*.xml for site description and menus.
- add your own web pages to wp-content/pages/, which are accessed by their filename in the url.
- If you want more customization to the look, modify the wp-content/themes/sensitive or create your own theme folder. Remember to set your theme in `wp-content/database/table_sitemeta.xml`, which contains all options for the site.

Usage 3: Adding new features to it

- Simply put your own files in `script/apps/WebServer/admin/` in your dev or working directory, your files will be loaded first before the one in `npl_package/main`.

Architecture

```
The architecture is based on Wordpress.org (4.0.1). Although everything is rewritten␣
↪in NPL, I have kept all functions, filters, and file names identical to wordpress.
See `wp-includes/` for the framework source code.
```

Code locations: * framework loader is in `wp-settings.page` * site options: `wp-content/database/table_sitemeta.xml`: such as theme, default menu, etc. * menus: `wp-content/database/table_nav_menu.xml`

Ajax framework

```
Any request url begins with `ajax/xxx` use the `wp-admin/admin-ajax.page`. it will␣
↪automatically load the page xxx.  and invoke `do_action('wp_ajax_xxx')`.
If the request url begins with `ajax/xxx?action=yyy`, then page xxx is loaded, and␣
↪`do_action('wp_ajax_xxx')` is invoked.
A page that handles ajax request needs to call `add_action('wp_ajax_xxx' function_
↪name)` to register a handler for ajax actions.
see `wp-content/pages/aboutus.page` for an example.
```

CHAPTER 56

Table Database

`Table database` is implemented in NPL and is the default and recommended database engine. It can be used without any external dependency or configuration.

Introduction to Table Database

A schema-less, server-less, NoSQL database able to process big data in multiple NPL threads, with extremly intuitive table-like API, automatic indexing and extremely fast searching, keeping data and algorithm in one system *just like how the human brain works*.

Local Storage API vs Full-Featured Database Solution

The raw table database has no configuration file. Please regard the raw table database as a local storage API provided by NPL, rather than a full-featured database solution. However its performance is good enough to be used as the database engine for medium sized projects. You need to write application-level code to have something like authentication, and data replication/hashing on multiple computers, etc. However, in future, some of these functions may be provided as additional libraries on top of the raw table database to provide enterprise level database solution that runs on multiple nodes.

Performance

Following is tested on my Intel-i7-3GHZ CPU. See test folder for test source code.

Run With Conservative Mode

Following is averaged value from 100000+ operations in a single thread

- Random non-index insert: `43478 inserts/second`
  - Async API tested with default configuration with 1 million records on a single thread.
- Round trip latency call:
  - Blocking API: `20000 query/s`
  - Non-blocking API: `11ms` or `85 query/s` (due to NPL time slice)

  - i.e. Round strip means start next operation after previous one is returned. This is latency test.
- Random indexed inserts: `17953 query/s`
  - i.e. start next operation immediately, without waiting for the first one to return.
- Random select with auto-index: `18761 query/s`
  - i.e. same as above, but with findOne operation.
- Randomly mixing CRUD operations: `965-7518` query/s
  - i.e. same as above, but randomly calling Create/Read/Update/Delete (CRUD) on the same auto-indexed table.
  - Mixing read/write can be slow when database grows bigger. e.g. you can get `18000 CRUD/s` for just 10000 records.

Run With Aggressive Mode

One can also use in-memory journal file or ignore OS disk write feedback to further increase DB throughput by 30-100% percent. See `Store.lua` for details. By default, this feature is off.

Code Examples:

```
    NPL.load("(gl)script/ide/System/Database/TableDatabase.lua");
    local TableDatabase = commonlib.gettable("System.Database.TableDatabase");
    -- this will start both db client and db server if not.
    local db = TableDatabase:new():connect("temp/mydatabase/", function() end);

    -- Note: `db.User` will automatically create the `User` collection table if not.
    -- clear all data
    db.User:makeEmpty({}, function(err, count) echo("deleted"..(count or 0)) end);
    -- insert 1
    db.User:insertOne(nil, {name="1", email="1@1",}, function(err, data)  assert(data.
↪email=="1@1")     end)
    -- insert 1 with duplicate name
    db.User:insertOne(nil, {name="1", email="1@1.dup",}, function(err, data)
↪assert(data.email=="1@1.dup")     end)

    -- find or findOne will automatically create index on `name` and `email` field.
    -- indices are NOT forced to be unique. The caller needs to ensure this see
↪`insertOne` below.
    db.User:find({name="1",}, function(err, rows) assert(#rows==2); end);
    db.User:find({name="1", email="1@1"}, function(err, rows) assert(rows[1].email==
↪"1@1"); end);
    -- find with compound index of name and email
    db.User:find({ ["+name+email"] = {"1", "1@1"} }, function(err, rows) assert(
↪#rows==1); end);

    -- force insert
    db.User:insertOne(nil, {name="LXZ", password="123"}, function(err, data)
↪assert(data.password=="123")  end)
    -- this is an update or insert command, if the query has result, it will actually
↪update first matching row rather than inserting one.
    -- this is usually a good way to force uniqueness on key or compound keys,
    db.User:insertOne({name="LXZ"}, {name="LXZ", password="1", email="lixizhi@yeah.net
↪"}, function(err, data)  assert(data.password=="1")   end)

    -- insert another one
```

```
    db.User:insertOne({name="LXZ2"}, {name="LXZ2", password="123", email=
↪"lixizhi@yeah.net"}, function(err, data)  assert(data.password=="123")     end)
    -- update one
    db.User:updateOne({name="LXZ2",}, {name="LXZ2", password="2", email="lixizhi@yeah.
↪net"}, function(err, data)  assert(data.password=="2") end)
    -- remove and update fields
    db.User:updateOne({name="LXZ2",}, {_unset = {"password"}, updated="with unset"},␣
↪function(err, data)  assert(data.password==nil and data.updated=="with unset") end)
    -- replace the entire document
    db.User:replaceOne({name="LXZ2",}, {name="LXZ2", email="lixizhi@yeah.net"},␣
↪function(err, data)  assert(data.updated==nil) end)
    -- force flush to disk, otherwise the db IO thread will do this at fixed interval
    db.User:flush({}, function(err, bFlushed) assert(bFlushed==true) end);
    -- select one, this will automatically create `name` index
    db.User:findOne({name="LXZ"}, function(err, user) assert(user.password=="1");  ␣
↪end)
    -- array field such as {"password", "1"} are additional checks, but does not use␣
↪index.
    db.User:findOne({name="LXZ", {"password", "1"}, {"email", "lixizhi@yeah.net"}},␣
↪function(err, user) assert(user.password=="1"); end)
    -- search on non-unqiue-indexed rows, this will create index `email` (not-unique␣
↪index)
    db.User:find({email="lixizhi@yeah.net"}, function(err, rows) assert(#rows==2);␣
↪end);
    -- search and filter result with password=="1"
    db.User:find({name="LXZ", email="lixizhi@yeah.net", {"password", "1"}, },␣
↪function(err, rows) assert(#rows==1 and rows[1].password=="1"); end);
    -- find all rows with custom timeout 1 second
    db.User:find({}, function(err, rows) assert(#rows==4); end, 1000);
    -- remove item
    db.User:deleteOne({name="LXZ2"}, function(err, count) assert(count==1); end);
    -- wait flush may take up to 3 seconds
    db.User:waitflush({}, function(err, bFlushed) assert(bFlushed==true) end);
    -- set cache to 2000KB
    db.User:exec({CacheSize=-2000}, function(err, data) end);
    -- run select command from Collection
    db.User:exec("Select * from Collection", function(err, rows) assert(#rows==3)␣
↪end);
    -- remove index fields
    db.User:removeIndex({"email", "name"}, function(err, bSucceed) assert(bSucceed ==␣
↪true) end)
    -- full table scan without using index by query with array items.
    db.User:find({ {"name", "LXZ"}, {"password", "1"} }, function(err, rows) assert(
↪#rows==1 and rows[1].name=="LXZ"); end);
    -- find with left subset of previously created compound key "+name+email"
    db.User:find({ ["+name"] = {"1", limit=2} }, function(err, rows) assert(#rows==2);
↪ end);
    -- return at most 1 row whose id is greater than -1
    db.User:find({ _id = { gt = -1, limit = 1, skip == 1} }, function(err, rows)␣
↪assert(#rows==1); echo("all tests succeed!") end);
```

Why A New Database System?

Current SQL/NoSQL implementation can not satisfy following requirements at the same time.

- Keeping data close to computation, much like our brain.

- Store arbitrary data without schema.
- Automatic indexing based on query usage.
- Provide both blocking/non-blocking API.
- Multithreaded architecture without using network connections for maximum local performance.
- Capable of storing hundreds of Giga Bytes of data locally.
- Native document storage format for NPL tables.
- Super easy client API just like manipulating standard NPL/lua tables.
- Easy to setup and deploy with NPL runtime.
- No server configuration, calling client API will automatically start the server on first use.

About Transactions

Each `write/insert` operation is by default a write command (virtual transaction). We will periodically (default is 50ms, see `Store.AutoFlushInterval`) flush all queued commands into disk. Everything during these period will either succeed or fail. If you are worried about data lose, you can manually invoke `flush` command, however doing so will greatly compromise performance. Please note `flush` command will affect the overall throughput of the entire DB system. In general, you can only get about `20-100` flush(real transactions) per second. Without enforcing transaction on each command, you can easily get a throughput of `6000` write commands per second (i.e. could be `100 times` faster).

There is one solution to get both high throughput and transaction.

The following solution give you ACID of a single command. After issuing a really important group of commands, and you want to ensure that these commands are actually successful like a transaction, the client can issue a `waitflush` command to check if the previous commands are successful. Please note that `waitflush` command may take up to 50ms or `Store.AutoFlushInterval` to return. You can make all calls asynchronous, so 99.99% times user get a fast feed back, but there is a very low chance that user may be informed of failure after 50ms. On client side, you may also need to prevent user to issue next transaction in 50ms,In most cases, users do not click mouse that quick, so this hidden logic goes barely noticed.

The limitation of the above approach is that you can only know whether a group of commands has been successfully flushed to disk, but you can not rollback your changes, nor can you know which commands fail and which are successful in case of multiple commands.

Opinion on Software Achitecture

Like the brain, we recommend that each computer manages its own chunk of data. As modern computers are having more computing cores, it is possible for a single (virtual) machine to manage 100-500GB of data locally or 1000 requests per second. Using a local database engine is the best choice in such situation for both performance and ease of deployment.

To scale up to even more data and requests, we devide data with machines and use higher level programming logics for communications. In this way, we control all the logics with our own code, rather than using general-purpose solutions like memcached, MangoDB(NoSQL), or SQLServer, etc.

Implementation Details

- Each data table(collections of data) is stored in a single sqlite database file.
- Each database file contains a indexed mapping from object-id to object-table(document).
- Each database file contains a schema table telling additional index keys used.
- Each database file contains a key to object-id table for each custom index key.
- All DB IO operations are performaned in a single dedicated NPL thread.

The above general idea can also be found in commercial database engine like CortexDB, see below.

User Guide

Because Table database is schema-less. It requires some caution of the programmer during development.

Auto Key Index

By default, when you are inserting records into the database, it will have only one internal unique key called `_id`. All subsequent query commands such as `findOne` or `find` or `insertOne`, `deleteOne`, `updateOne` command with query fields, such as `name` will automatically add a new index key with corresponding name such as `name`, and rebuild the index table for all existing records. Please note, all indices are NOT-constraint to be unique, it is up to the caller to ensure index usage. Different records with the same key value are all stored in the index table.

> One can also force not-to-use index when querying by using array items rather than key,value pairs in the query parameter, see below.

```
    NPL.load("(gl)script/ide/System/Database/TableDatabase.lua");
    local TableDatabase = commonlib.gettable("System.Database.TableDatabase");
    -- this will start both db client and db server if not.
    local db = TableDatabase:new():connect("temp/mydatabase/", function() end);

    -- Note: `db.User` will automatically create the `User` collection table if not.
    -- clear all data
    db.User:makeEmpty({}, function(err, count) echo("deleted"..(count or 0)) end);
    -- insert 1
    db.User:insertOne(nil, {name="1", email="1@1",}, function(err, data)  echo(data)
↪  end)
    -- insert 1 with duplicate name
    db.User:insertOne(nil, {name="1", email="1@1.dup",}, function(err, data)
↪echo(data)    end)

    -- find or findOne will automatically create index on `name` and `email` field.
    -- indices are NOT forced to be unique. The caller needs to ensure this see
↪`insertOne` below.
    -- this will return two records using one index table `name`
    db.User:find({name="1",}, function(err, rows) echo(rows); end);
    -- this will return one record using two index table `name` and `email`
    db.User:find({name="1", email="1@1"}, function(err, rows) echo(rows); end);

    -- force insert, insertOne with no query parameter will force insertion.
    db.User:insertOne(nil, {name="LXZ", password="123"}, function(err, data)
↪echo(data)    end)
    -- this is an update or insert command, if the query has result, it will actually
↪update first matching row rather than inserting one.
```

```
    -- this is usually a good way to force uniqueness on key or compound keys
    -- so the following will update existing record rather than inserting a new one
    db.User:insertOne({name="LXZ"}, {name="LXZ", password="1", email="lixizhi@yeah.net
↪"}, function(err, data)  echo(data)   end)
```

All query fields in `findOne`, `updateOne`, etc can contain array items, which performs additional filter checks without automatically create index, like below. `name` is an indexed key, where as `password` and `email` are not, so they need be specified as array items in the query parameter.

```
    -- array fields such as {"password", "1"} are additional checks, but does not use␣
↪index.
    db.User:findOne({name="LXZ", {"password", "1"}, {"email", "lixizhi@yeah.net"}},␣
↪function(err, user) echo(user); end)
```

In case, you make a mistake with index during development, please use `DB manager` in NPL code wiki to remove any index that you accidentally create. or simply call `removeIndex` to remove all or given index keys.

Force Key Uniqueness and `Upsert`

If you want to force uniqueness on a given key, one can specify the unique key in query field when inserting a new record, which turns the `insertOne` command into an `Upsert` command. See below:

```
-- this is an update or insert command, if the query has result, it will actually␣
↪update first matching row rather than inserting one.
-- this is usually a good way to force uniqueness on key or compound keys
-- so the following will update existing record rather than inserting a new one
db.User:insertOne({name="LXZ"}, {name="LXZ", password="1", email="lixizhi@yeah.net"},␣
↪function(err, data)  echo(data)   end)
```

Some notes:

- Query commands such as `find`, `findOne`, `insertOne` will automatically create index for fields in query parameter, unless field is specified in array format.
- Query with mixed index and non-index is done by fetching using index first and then filter result with non-indexed fields. If there is no indexed fields, then non-indexed query can be very slow, since it will linearly search on all records, which can be very slow.
- Query with multiple fields is supported. It will first fetch ids using each index field, and calculate the shared ids and then fetch these rows in a batch.

Write-Ahead-Log and Checkpointing

By default, TableDatabase uses write-ahead-log. We will flush data to WAL file every 50ms; and we will do WAL checkpoint every 5 seconds or 1000 dirty pages. One can configure these two intervals, like below. However, it is not recommended to change these values.

```
NPL.load("(gl)script/ide/System/Database/Store.lua");
local Store = commonlib.gettable("System.Database.Store");
-- We will wait for this many milliseconds when meeting the first non-queued command␣
↪before commiting to disk. So if there are many commits in quick succession, it will␣
↪not be IO bound.
Store.AutoFlushInterval = 50;
Store.AutoCheckPointInterval = 5000;
```

Checkpointing may take 10ms-1000ms depending on usage, during which time it will block all write operations, so one may experience a latency in API once in a while.

Message Queue Size

One can set the message queue size for both the calling thread and db processor thread. The default value is good enough for most situations.

```
db.name:exec({QueueSize=10001});
```

Memory Cache Size

Change the suggested maximum number of database disk pages that TableDatabase will hold in memory at once per open database file. The default suggested cache size is -2000, which means the cache size is limited to 2048KB of memory. You can set it to a much bigger value, since it is only allocated on demand. Negative value is KB, Positive is number of pages, each page is 4KB.

```
db.name:exec({CacheSize=-2000});
```

Bulk Operations and Message Deliver Guarantee

Table database usually has a high throughout such as 5000-40000 requests/sec. However, if you are doing bulk insertion of millions of records. You should do chunk by chunk using callbacks, otherwise you will either reach the limit of NPL thread buffer or TCP socket buffer(in case it is remote process). A good way is to keep at most 1000 active requests at any one time (`System.Concurrency` has helper class for this kind of jobs). This way you can get a good throughput with message delivery guarantees.

Following is example of inserting 1 million records (takes about 23 seconds). You can paste following code to NPL Code Wiki's console to test.

```
    NPL.load("(gl)script/ide/System/Database/TableDatabase.lua");
    local TableDatabase = commonlib.gettable("System.Database.TableDatabase");

        -- this will start both db client and db server if not.
    local db = TableDatabase:new():connect("temp/mydatabase/");

    db.insertNoIndex:makeEmpty({});
    db.insertNoIndex:flush({});

    NPL.load("(gl)script/ide/Debugger/NPLProfiler.lua");
    local npl_profiler = commonlib.gettable("commonlib.npl_profiler");
    npl_profiler.perf_reset();

    npl_profiler.perf_begin("tableDB_BlockingAPILatency", true)
    local total_times = 1000000; -- a million non-indexed insert operation
    local max_jobs = 1000; -- concurrent jobs count
    NPL.load("(gl)script/ide/System/Concurrent/Parallel.lua");
    local Parallel = commonlib.gettable("System.Concurrent.Parallel");
    local p = Parallel:new():init()
    p:RunManyTimes(function(count)
        db.insertNoIndex:insertOne(nil, {count=count, data=math.random()},␣
↪function(err, data)
            if(err) then
```

```
                echo({err, data});
            end
            p:Next();
        end)
    end, total_times, max_jobs):OnFinished(function(total)
        npl_profiler.perf_end("tableDB_BlockingAPILatency", true)
        log(commonlib.serialize(npl_profiler.perf_get(), true));
    end);
```

In future version, we may support build-in Bulk API.

Async vs Sync API

Table database provides both sync and asynchronous API, and they can be use simultaneously. However, sync interface is disabled by default. One has to manually enable it, such as during initialization. In sync interface, if you do not provide a callback function, then the API block until result is returned, otherwise the API return AsyncTask object immediately. See following example:

```
-- enable sync mode once and for all in current thread.
db:EnableSyncMode(true);
-- synchronous call will block until data is fetched.
local err, data = db.User:insertOne(nil, {name="LXZ2", email="sync mode"})
-- async call return a task object immediately without waiting result.
local task = db.User:insertOne(nil, {name="LXZ2", email="sync mode"}, function(err,␣
↪data) end)
```

Synchronous call will pause the entire NPL thread, and is `NOT` the recommended to use, that is why we disabled this feature by default.

Instead, during web development, there is another way to use async API in synchronous way, which is to use coroutine. In NPL web server, we provide `resume/yield`. See NPLServerPage's mixing sync/async code section.

Ranged Query and Pagination

Paging through your data is one of the most common operations with TableDB. A typical scenario is the need to display your results in chunks in your UI. If you are batch processing your data it is also important to get your paging strategy correct so that your data processing can scale.

Let's walk through an example to see the different ways of paging through data in TableDB. In this example, we have a database of user data that we need to page through and display 10 users at a time. So in effect, our page size is 10. Let us first create our db with 10000 users of scheme `{_id,name,company,state}`.

```
    NPL.load("(gl)script/ide/System/Database/TableDatabase.lua");
    local TableDatabase = commonlib.gettable("System.Database.TableDatabase");
    local db = TableDatabase:new():connect("temp/mydatabase/");

    -- add some data
    for i=1, 10000 do
        db.pagedUsers:insertOne({name="name"..i},
            {name="name"..i, company="company"..i, state="state"..i}, function() end)
    end
```

Approach One: Use `limit` and `skip`

TableDB support `limit` and `offset|skip` key words in query parameter on an indexed field or the internal `_id` field. `offset or skip` tells TableDB to skip some items before returning at most `limit` number of items. `gt` means `greater than` to select a given range of data.

```
    -- page 1
    db.pagedUsers:find({_id = {gt=-1, limit=10, skip=0}}, function(err, users) ␣
↪echo(users) end);
    -- page 2
    db.pagedUsers:find({_id = {gt=-1, limit=10, skip=10}}, function(err, users) ␣
↪echo(users) end);
    -- page 3
    db.pagedUsers:find({_id = {gt=-1, limit=10, skip=20}}, function(err, users) ␣
↪echo(users) end);
```

You get the idea. In general to retrieve page `n` the code looks like this

```
    local pagesize = 10; local n = 10;
    db.pagedUsers:find({_id = {gt=-1, limit=pagesize, skip=(n-1)*pagesize}},␣
↪function(err, users)  echo(users) end);
```

However as the `offset` of your data increases this approach has serious performance problems. The reason is that every time the query is executed, the server has to walk from the beginning of the first non-skipped row to the specified offset. As your offset increases this process gets slower and slower. Also this process does not make efficient use of the indexes. So typically the `skip` and `limit` approach is useful when you have small `offset`. If you are working with large data sets with very big `offset` values you need to consider other approaches.

Approach Two: Using `find` and `limit`

The reason the previous approach does not scale very well is the `skip` command. So the goal in this section is to implement paging without using the `skip` command. For this we are going to leverage the natural order in the stored data like a time stamp or an id stored in the document. In this example we are going to use the `_id` stored in each document. `_id` is a TableDB auto-increased internal id.

```
    -- page 1
    db.pagedUsers:find({_id = { gt = -1, limit=10}}, function(err, users)
        echo(users)
        -- Find the id of the last document in this page
        local last_id = users[#users]._id;

        -- page 2
        db.pagedUsers:find({_id = { gt = last_id, limit=10}}, function(err, users)
            echo(users)
            -- Find the id of the last document in this page
            local last_id = users[#users]._id;
            -- page 3
            -- ...
        end);
    end);
```

Notice how the `gt` keywords to select rows whose values is `greater than` the specified value and sort data in ascending order. One can do this for any indexed field, either it is number or string, see below for ranged query in TableDB:

```
    NPL.load("(gl)script/ide/System/Database/TableDatabase.lua");
    local TableDatabase = commonlib.gettable("System.Database.TableDatabase");

    -- this will start both db client and db server if not.
    local db = TableDatabase:new():connect("temp/mydatabase/");

    -- add some data
    for i=1, 100 do
        db.rangedTest:insertOne({i=i}, {i=i, data="data"..i}, function() end)
    end

    -- return at most 5 records with i > 90, skipping 2. result in ascending order
    db.rangedTest:find({ i = { gt = 95, limit = 5, offset=2} }, function(err, rows)
        echo(rows); --> 98,99,100
    end);

    -- return at most 20 records with _id > 98, result in ascending order
    db.rangedTest:find({ _id = { gt = 98, limit = 20} }, function(err, rows)
        echo(rows); --> 99,100
    end);

    -- do a full table scan without using index
    db.rangedTest:find({ {"i", { gt = 55, limit=2, offset=1} }, {"data", {lt="data60"}
↪ }, }, function(err, rows)
        echo(rows); --> 57,58
    end);

    db.rangedTest:find({ {"i", { gt = 55} }, {"data", "data60"}, }, function(err,␣
↪rows)
        echo(rows); --> 60
    end);
```

Please note that array fields in query object will not auto-create or use any index, instead a full table scan is performed which can be slow. Ranged query is also supported in array fields, but it is implemented via full table scan.

Compound key is more suitable for ranged query see Compound keys section below for how to use it.

Count Record

Rule of thumb: avoid using `count`!

Implementation of full paging usually requires you to get the count of all items in a database. However, it is not trivial to get accurate count. SQL query like `select count(*) as count from Collection` may take seconds or even minutes to return for tables with many rows like 100 millions. This is because almost every SQL engine uses B-tree, but B-tree does not store count. The same story is true for both TableDB, MySQL, etc.

For a faster count, you can create a counter table and let your application update it according to the inserts and deletes it does. Or you may not allow the user to scroll to the last page, but show more pages progressively.

If you do not mind the performance or your table size is small, TableDB provides the `count` query

```
    NPL.load("(gl)script/ide/System/Database/TableDatabase.lua");
    local TableDatabase = commonlib.gettable("System.Database.TableDatabase");
    local db = TableDatabase:new():connect("temp/mydatabase/");

    db.countTest:removeIndex({}, function(err, bRemoved) end);
```

```
    -- compound keys
    for i=1, 100 do
        db.countTest:insertOne({name="name"..i}, {
            name="name"..i,
            company = (i%2 == 0) and "tatfook" or "paraengine",
            state = (i%3 == 0) and "china" or "usa"}, function() end)
    end

    -- count all rows
    db.countTest:count({}, function(err, count)
        assert(count == 100)
    end);
    -- count with compound keys
    db.countTest:count({["+company"] = {"tatfook"}}, function(err, count)
        assert(count == 50)
    end);
    -- count with complex query
    db.countTest:count({["+state+name+company"] = {"china", gt="name50"}},␣
↪function(err, count)
        assert(count == 19)
    end);
```

Compound Indices

To efficiently query data with multiple key values, one needs to use compound indices. Please note a compound key of (key1, key2, key3) can also be used to query (key1, key2) and (key1), so the order of subkeys are very important.

If you created a compound index say on (key1, key2, key3), then only the right most key in the query can contain ranged constraint, all other keys to its left such as (key1 and key2) must be present (no gaps) and specified in equal constraint.

To understand compound indices, one needs to understand how index works in standard relational database engine, because a TableDB query usually use only one internal index to do all the complex queries and an index is a B-tree. So compound index can not be used where B-tree is not capable of. Read query planner of sqlite here for more details.

The syntax of using compound key is by prepending "+|-" to one or more field names, such as "+key1+key2+key3", there can be at most 4 sub keys, see below for examples. "+" means ascending order, "-" means descending order.

Examples:

```
    NPL.load("(gl)script/ide/System/Database/TableDatabase.lua");
    local TableDatabase = commonlib.gettable("System.Database.TableDatabase");
    local db = TableDatabase:new():connect("temp/mydatabase/");

    db.compoundTest:removeIndex({}, function(err, bRemoved) end);

    -- compound keys
    for i=1, 100 do
        db.compoundTest:insertOne({name="name"..i}, {
            name="name"..i,
            company = (i%2 == 0) and "tatfook" or "paraengine",
            state = (i%3 == 0) and "china" or "usa"}, function() end)
    end

    -- compound key can also be created on single field "+company". it differs from␣
↪standard key "company".
    db.compoundTest:find({["+company"] = {"paraengine", limit=5, skip=3}},␣
↪function(err, users)
```

```
        assert(#users == 5)
    end);

    -- create a compound key on +company+name (only the right most key can be paged)
    -- `+` means index should use ascending order, `-` means descending. such as
↪"+company-name"
    -- it will remove "+company" key, since the new component key already contains␣
↪the "+company".
    db.compoundTest:find({["+company+name"] = {"tatfook", gt="", limit=5, skip=3}},␣
↪function(err, users)
        assert(#users == 5)
    end);

    -- a query of exact same left keys after a compound key is created will␣
↪automatically use
    -- the previously created compound key, so "+company",  "+company+name" in␣
↪following query are the same.
    db.compoundTest:find({["+company"] = {"tatfook", limit=5, skip=3}}, function(err,␣
↪users)
        assert(#users == 5)
    end);

    -- the order of key names in compound index is important. for example:
    -- "+company+state+name" shares the same compound index with "+company" and
↪"+company+state"
    -- "+state+name+company" shares the same compound index with "+state" and
↪"+state+name"
    db.compoundTest:find({["+state+name+company"] = {"china", gt="name50", limit=5,␣
↪skip=3}}, function(err, users)
        assert(#users == 5)
    end);

    -- compound keys with descending order. Notice the "-" sign before `name`.
    db.compoundTest:find({["+state-name+company"] = {"china", limit=5, skip=3}},␣
↪function(err, users)
        assert(#users == 5)
    end);

    -- updateOne with compound key
    db.compoundTest:updateOne({["+state-name+company"] = {"usa", "name1", "paraengine
↪"}}, {name="name0_modified"}, function(err, user)
        assert(user.name == "name0_modified")
    end);

    -- this query is ineffient since it uses intersection of single keys (with many␣
↪duplications).
    -- one should consider use "+state+company", instead of this.
    db.compoundTest:find({state="china", company="tatfook"}, function(err, users)
        assert(#users == 16)
    end);

    -- this query is ineffient since it uses intersection of single keys.
    -- one should consider use "+state+company", instead of this.
    db.compoundTest:count({state="china", company="tatfook"}, function(err, count)
        assert(count == 16)
    end);
```

Compound key vs Standard key

Compound key is very different from standard key in terms of internal implementations. They can coexist in the same table even for the same key. For example:

```
-- query with standard key "name"
db.User:find({name="1",});
-- query with compound key "+name"
db.User:find({["+name"]="1",});
```

The above code will create two different kinds of internal index tables for the "name" field.

Compound key index table is more similar to traditional relational database table. For example, if one creates a compound key of "+key1+key2+key3", then an internal index table of `(cid UNIQUE,name1,name2,name3)` is created with composite index of `CREATE INDEX (name1,name2,name3)`. For each row in the collection, there is a matching row in its internal index table.

However, a standard key in tableDB stores index table differently as `(name UNIQUE,cids)`, where cids is a text array of collection `cid`. There maybe far less rows in the internal index table than the actual collection.

Both keys have advantages and disadvantages, generally speaking:

- standard key is faster and good for index intersections. Most suitable for keys with 1-1000 duplications.
- compound key is good for sorting and ranged query. Suitable for keys with any duplicated rows.

The biggest factor that you should use a compound key is that your key's values have many duplications. For example, if you want to query with `gender+age` with millions of `user` records, compound key is the only way to go.

For other single key situations, standard key is recommended. For example, with standard keys you can get all `projects` that belongs to a given `userid` very efficiently, given that each user has far less than 1000 projects.

Database Viewer Tools

NPLCodeWiki contains a `db manager` in its tools menu, which can be used to view and modify data, as well as examine and removing `table indices`. It is very important to examine and remove unnecessary indices due to coding mistakes.

Remove Table Fields

To remove table fields, use `updateOne` with `_unset` query parameter like below.

```
    -- remove "password" field from the given row
    db.User:updateOne({name="LXZ2",}, {_unset = {"password"}, updated="with unset"},
       function(err, data)
          assert(data.password==nil and data.updated=="with unset")
    end)
```

Another way is the use `replaceOne` to replace the entire document.

```
-- replace the entire document
db.User:replaceOne({name="LXZ2",}, {name="LXZ2", email="lixizhi@yeah.net"},␣
↪function(err, data)  assert(data.updated==nil) end)
```

Always Use Models

In typical web applications, we use model/view/controller (or MVC) architecture. A model is usually a wrapper of a single schema-less database record in TableDatabase. Usually a model should provide CRUD(create/read/update/delete) operations according to user-defined query. It is the model's responsibility to ensure input/output are valid, such as whether a string is too long, or whether the caller is querying with a non-indexed key.

In case, you are using client side controllers, one can use a router page to automatically redirect URL AJAX queries to the model's function calls.

For a complete example of url redirection and model class, please see `script/apps/WebServer/admin/wp-content/pages/wiki/routes.page` and `script/apps/WebServer/admin/wp-content/pages/models/abstract/base.page`

Following code illustrates the basic ideas.

Router page such as in `routes.page`

```
        local model = models and models[modelname];
        if(not model) then
            return response:status(404):send({message="model not found"});
        else
            model = model:new();
        end

        -- redirect CRUB URL to method in model.
        if(request:GetMethod() == "GET") then
            if(model.get) then
                local result = model:get(request:getparams());
                return response:send(result);
            end
        elseif(request:GetMethod() == "PUT") then
            if(params and params:match("^new")) then
                if(model.create) then
                    local result = model:create(request:getparams());
                    return response:send(result);
                end
            else
                if(model.update) then
                    local result = model:update(request:getparams());
                    return response:send(result);
                end
            end
        elseif(request:GetMethod() == "DELETE") then
            if(model.delete) then
                local result = model:delete(request:getparams());
                return response:send(result);
            end
        end
```

And in model class, such as `models/site.page`

```
<?npl
--[[
Title: a single web site of a user
Author: LiXizhi
Date: 2016/6/28
]]
```

```
include_once("base.page");

local site = inherit(models.base, gettable("models.site"));

site.db_name = "site";

function site:ctor()
    -- unique name of the website, usually same as owner name
    self:addfield("name", "string", true, 30);
    -- markdown text description
    self:addfield("desc", "string");
    -- such as "https://github.com/LiXizhi/wiki"
    self:addfield("store", "string", false, 200);
    -- owner name, not unique
    self:addfield("owner", "string", false, 30);
end
```

Data Replication

TableDB is a fully ACID and fast database (it uses Sqlite as its internal storage engine). We are going to support tableDB very actively as an integral part of NPL. NPL language itself is designed for parallelism, the current version of tableDB is still a local SQL engine. It is hence not hard to use NPL to leverage TableDB to a fully distributed and fault-tolerant database system.

TODO: RAFT implementation coming in 2017

https://raft.github.io/ is a concensus algorithm that we are going to implement using NPL and TableDB itself to leverage TableDB to support `master/slave` data replication for high-availability database systems.

There are many existing distributed SQL server that are built over similar architecture and use Sqlite as the storage engine, like these

> We highly recommend the native database system in NPL: TableDatabase. It is way faster and built-in with NPL runtime.

CHAPTER 57

Using MySql Client

Install Guide

- On linux platform, mysql client is by default installed when you build NPLRuntime from source code. You should find a `libluasql.so` in NPL runtime directory.
- On windows platform, you must manually install the mysql client connector plugin.

  https://github.com/LiXizhi/luasql

Once installed, you can use it like below:

```
NPL.load("(gl)script/ide/mysql/mysql.lua");
local MySql = commonlib.gettable("System.Database.MySql");
local mysql_db = MySql:new():init(db_user, db_password, db_name, db_host, db_port);
```

A Better Way to Code

It is recommended that you write a wrapper file like `my_db.page` which exposes a global object such as `my_db` that contains functions to access to your actual database.

An example in the Admin web site framework is here.

- `script/apps/WebServer/admin/wp-includes/wp-db.page`

It will manage connection strings (passwords, ips, pools) from global configuration files, etc. In all other server pages, you simply include the file, and call its functions like this

```
local query = string.format("select * from wp_users where %s = '%s' ",db_field,value);
local user = wpdb:get_row(query);
```

CHAPTER 58

Deploy NPL Web Server With SSL (https)

This post shows `How To Configure Nginx with SSL as a Reverse Proxy for NPL Web Server`.

Why NPL Does Not Support SSL Natively?

NPL protocol is TCP based, which can run HTTP and NPL's TCP protocol on the `same` port. If the TCP connection is encoded with SSL, it will break NPL's internal TCP protocol. This is why NPL does not support SSL natively. However, NPL server can fetch data via `https`, so it is perfectly fine to call SSL protected rest API within the NPL server.

Introduction

By default, NPL comes with its own built in web server, which listens on port 8099. This is convenient if you run a private NPL instance, or if you just need to get something up quickly and don't care about security. Once you have real production data going to your host, though, it's a good idea to use a more secure web server proxy in front like Nginx.

Prerequisites

This post will detail how to wrap your site with SSL using the Nginx web server as a reverse proxy for your NPL instance. This tutorial assumes some familiarity with Linux commands, a working NPL Runtime installation, and a Ubuntu 14.04 installation.

You can install NPL runtime later in this tutorial, if you don't have it installed yet.

Step One — Configure Nginx

Nginx has become a favored web server for its speed and flexibility in recents years, so that is the web server we will be using.

Install Nginx

Update your package lists and install Nginx:

```
sudo apt-get update
sudo apt-get install nginx
```

Get a Certificate

Next, you will need to purchase or create an SSL certificate. These commands are for a self-signed certificate, but you should get an officially signed certificate if you want to avoid browser warnings.

Move into the proper directory and generate a certificate:

```
cd /etc/nginx
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /etc/nginx/cert.key -
↪out /etc/nginx/cert.crt
```

You will be prompted to enter some information about the certificate. You can fill this out however you'd like; just be aware the information will be visible in the certificate properties. We've set the number of bits to 2048 since that's the minimum needed to get it signed by a CA. If you want to get the certificate signed, you will need to create a CSR.

Edit the Configuration

Next you will need to edit the default Nginx configuration file.

```
sudo nano /etc/nginx/sites-enabled/default
```

Here is what the final config might look like; the sections are broken down and briefly explained below. You can update or replace the existing config file, although you may want to make a quick copy first.

```
server {
    listen 80;
    return 301 https://$host$request_uri;
}

server {

    listen 443;
    server_name npl.domain.com;

    ssl_certificate           /etc/nginx/cert.crt;
    ssl_certificate_key       /etc/nginx/cert.key;

    ssl on;
    ssl_session_cache  builtin:1000  shared:SSL:10m;
    ssl_protocols  TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers HIGH:!aNULL:!eNULL:!EXPORT:!CAMELLIA:!DES:!MD5:!PSK:!RC4;
    ssl_prefer_server_ciphers on;

    access_log            /var/log/nginx/npl.access.log;

    location / {

      proxy_set_header        Host $host;
```

```
      proxy_set_header        X-Real-IP $remote_addr;
      proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header        X-Forwarded-Proto $scheme;

      # Fix the "It appears that your reverse proxy set up is broken" error.
      proxy_pass          http://localhost:8099;
      proxy_read_timeout  90;

      proxy_redirect      http://localhost:8099 https://npl.domain.com;
    }
  }
```

In our configuration, the cert.crt and cert.key settings reflect the location where we created our SSL certificate. You will need to update the servername and `proxyredirect` lines with your own domain name. There is some additional Nginx magic going on as well that tells requests to be read by Nginx and rewritten on the response side to ensure the reverse proxy is working.

The first section tells the Nginx server to listen to any requests that come in on port 80 (default HTTP) and redirect them to HTTPS.

```
...
server {
   listen 80;
   return 301 https://$host$request_uri;
}
...
```

Next we have the SSL settings. This is a good set of defaults but can definitely be expanded on. For more explanation, please read this tutorial.

```
...
  listen 443;
  server_name npl.domain.com;

  ssl_certificate           /etc/nginx/cert.crt;
  ssl_certificate_key       /etc/nginx/cert.key;

  ssl on;
  ssl_session_cache  builtin:1000  shared:SSL:10m;
  ssl_protocols  TLSv1 TLSv1.1 TLSv1.2;
  ssl_ciphers HIGH:!aNULL:!eNULL:!EXPORT:!CAMELLIA:!DES:!MD5:!PSK:!RC4;
  ssl_prefer_server_ciphers on;
  ...
```

The final section is where the proxying happens. It basically takes any incoming requests and proxies them to the NPL instance that is bound/listening to port 8099 on the local network interface. This is a slightly different situation, but this tutorial has some good information about the Nginx proxy settings.

```
...
location / {

    proxy_set_header        Host $host;
    proxy_set_header        X-Real-IP $remote_addr;
    proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header        X-Forwarded-Proto $scheme;

    # Fix the "It appears that your reverse proxy set up is broken" error.
    proxy_pass          http://localhost:8099;
```

```
    proxy_read_timeout  90;

    proxy_redirect      http://localhost:8099 https://npl.domain.com;
}
...
```

A few quick things to point out here. If you don't have a domain name that resolves to your NPL server, then the proxy_redirect statement above won't function correctly without modification, so keep that in mind.

Step Two — Configure NPL Runtime

As stated previously, this tutorial assumes that NPL is already installed. This tutorial will show you how to install NPL if necessary. You will probably need to switch to the root user for that article.

For NPL to work with Nginx, we need to update the NPL config to listen only on the localhost interface instead of all (0.0.0.0), to ensure traffic gets handled properly. This is an important step because if NPL is still listening on all interfaces, then it will still potentially be accessible via its original port (8099).

For example, one may start npl webserver like this

```
pkill -9 npl
npl -d root="WebServerExample/" ip="127.0.0.1" port="8099" bootstrapper="script/apps/
↪WebServer/WebServer.lua"
```

Notice that the `ip="127.0.0.1"` setting needs to be either added or modified.

Then go ahead and restart NPL and Nginx.

```
sudo service nginx restart
```

You should now be able to visit your domain using either HTTP or HTTPS, and the NPL web site will be served securely. You will see a certificate warning if you used a self-signed certificate.

Conclusion

The only thing left to do is verify that everything worked correctly. As mentioned above, you should now be able to browse to your newly configured URL - npl.domain.com - over either HTTP or HTTPS. You should be redirected to the secure site, and should see some site information, including your newly updated SSL settings. As noted previously, if you are not using hostnames via DNS, then your redirection may not work as desired. In that case, you will need to modify the proxy_pass section in the Nginx config file.

You may also want to use your browser to examine your certificate. You should be able to click the lock to look at the certificate properties from within your browser.

References

This is a rewrite of this post in terms of NPL.

CHAPTER 59

Use Links

Source code

Development

Documentation

Plugins and Mod

CHAPTER 60

NPL Runtime Performance Compare

High-Performance JIT Compiler

NPL syntax is 100% compatible with Lua, therefore it can be configured to utilize JIT compiler (Luajit). In general, luajit is believed to be one of the fastest JIT compiler in the world. It's speed is close to C/C++, and can even outperform static typed languages like java/C#. It is the fastest dynamic language in the world. However, special care needs to be taken when writing test cases, since badly written test case can make the same code 100 times slower.

Compare Chart

Following is from Julia. Source code: C, Fortran, Python, Matlab/Octave, R, JavaScript, Java, Go, Lua.

The following micro-benchmark results were obtained on a single core (serial execution) on an Intel(R) Xeon(R) CPU E7-8850 2.00GHz CPU with 1TB of 1067MHz DDR3 RAM, running Linux:

| |C/Fortran |Python |**NPL** |Java|JavaScript|Go |R |Matlab |Octave|————————|———-|———-|———-|———-|———-|———-|———-|———-|———- | |gcc5.1 |3.4.3 | 1.0 |1.8 |V8 |go1.5 |3.2.2 |R2015b |4.0.0|mandel |0.81 |15.32 |**0.67**|1.35|0.66 |1.11 |53.16 |7.58 |451.81|fib |0.70 |77.76 |**1.71**|1.21|3.36 |1.86 |533.52 |26.89 |9324.35 |rand_mat_mul |3.48 |1.14 |**1.16**|2.36|15.07|1.42 |1.57 |1.12 |1.12|rand_mat_stat |1.45 |17.93 |**3.27**|3.92|2.30 |2.96 |14.56 |14.52 |30.93|parse_int |5.05 |17.02 |**5.77**|3.35|6.06 |1.20 |45.73 |802.52 |9581.44 |quicksort |1.31 |32.89 |**2.03**|2.60|2.70 |1.29 |264.54 |4.92 |1866.01 |pi_sum |1.00 |21.99 |**1.00**|1.00|1.01 |1.00 |9.56 |1.00 |299.31

Figure: benchmark times relative to C (smaller is better, C performance = 1.0).

> C and Fortran compiled by gcc 5.1.1, taking best timing from all optimization levels (-O0 through -O3). C, Fortran, Go. Python 3 was installed from the Anaconda distribution. The Python implementations of rand_mat_stat and rand_mat_mul use NumPy (v1.9.2) functions; the rest are pure Python implementations. Benchmarks can also be seen here as a plot created with Gadfly.

These benchmarks, while not comprehensive, do test compiler performance on a range of common code patterns, such as function calls, string parsing, sorting, numerical loops, random number generation, and array operations. It is important to note that these benchmark implementations are not written for absolute maximal performance (the fastest code to compute fib(20) is the constant literal 6765). Rather, all of the benchmarks are written to test the performance of specific algorithms implemented in each language. In particular, all languages use the same algorithm: the Fibonacci benchmarks are all recursive while the pi summation benchmarks are all iterative; the "algorithm" for random matrix multiplication is to call LAPACK, except where that's not possible, such as in JavaScript. The point of these benchmarks is to compare the performance of specific algorithms across language implementations, not to compare the fastest means of computing a result, which in most high-level languages relies on calling C code.

NPL Database Performance

More information, please see UsingTableDatabase

Following is tested on my Intel-i7-3GHZ CPU. See test folder for test source code.

Run With Conservative Mode

Following is averaged value from 100000+ operations in a single thread

- Random non-index insert: `43478 inserts/second`
  - Async API tested with default configuration with 1 million records on a single thread.
- Round trip latency call:
  - Blocking API: `20000 query/s`
  - Non-blocking API: `11ms` or `85 query/s` (due to NPL time slice)
  - i.e. Round strip means start next operation after previous one is returned. This is latency test.
- Random indexed inserts: `17953 query/s`
  - i.e. start next operation immediately, without waiting for the first one to return.
- Random select with auto-index: `18761 query/s`
  - i.e. same as above, but with findOne operation.
- Randomly mixing CRUD operations: `965-7518` query/s
  - i.e. same as above, but randomly calling Create/Read/Update/Delete (CRUD) on the same auto-indexed table.
  - Mixing read/write can be slow when database grows bigger. e.g. you can get `18000 CRUD/s` for just 10000 records.

NPL Web Server Performance

The following is done using ApacheBench (AB tool) with 5000 requests and 1000 concurrency on a 1 CPU 1GB memory virtual machine. The queried content is http://paracraft.wiki/, which is a fairly standard dynamic web page.

```
root@iZ62yqw316fZ:/opt/paracraftwiki# ab -n 5000 -c 1000 http://paracraft.wiki/
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking paracraft.wiki (be patient).....done

Finished 5000 requests

Server Software:        NPL/1.1
Server Hostname:        paracraft.wiki
Server Port:            80

Document Path:          /
Document Length:        24736 bytes
```

```
Concurrency Level:      1000
Time taken for tests:   3.005 seconds
Complete requests:      5000
Failed requests:        0
Write errors:           0
Total transferred:      124730000 bytes
HTML transferred:       123680000 bytes
Requests per second:    1664.05 [#/sec] (mean)
Time per request:       600.944 [ms] (mean)
Time per request:       0.601 [ms] (mean, across all concurrent requests)
Transfer rate:          40538.43 [Kbytes/sec] received
```

References

There are a few benchmark compare sites:

- https://github.com/attractivechaos/plb: algorithm compare
- TechEmpower: compare web server framework only
- Luajit Performance
- Computer benchmark game: No luajit.

CHAPTER 61

ParaEngine

class `Nutshell`
With a little bit of a elaboration, should you feel it necessary.

Public Types

enum `Tool`
Our tool set.

The various tools we can opt to use to crack this particular nut

Values:

`kHammer` = 0
What? It does the job.

`kNutCrackers`
Boring.

`kNinjaThrowingStars`
Stealthy.

Public Functions

`Nutshell`()
Nutshell constructor.

`~Nutshell`()
Nutshell destructor.

void **`crack`**(*Tool tool*)
Crack that shell with specified tool

Parameters

- `tool`: - the tool with which to crack the nut

bool **`isCracked`**() **Return**
Whether or not the nut is cracked

CHAPTER 62

ParaScripting

N