Monk-C

a toolkit for oop by C language

Version 0.5

Contents:

- 1. Overview
  - 1.1 Software Layers of Monk-C oop toolkit
  - 1.2 Data structure for oop abstract
  - 1.3 The power of Hash algorithm
  - 1.4 Method resolve
  - 1.5 Mode for objects
  - 1.6 Reference counting for memory management
  - 1.7 Floating point data
- 2. Syntax
  - 2.1 Syntax for class define
  - 2.2 Syntax for class implement
  - 2.3 Syntax for protocol
  - 2.4 Syntax for mode
  - 2.5 Syntax for reference counting
  - 2.6 Syntax for floating point data
- 3. Practice and Debug
  - 3.1 The Makefile
  - 3.2 Introduce of log system
  - 3.3 Traps of Monk-C
  - 3.4 Pay attention to memory management
- 4. Concurrency
  - 4.1 Compare and swap algorithm for lock free implementation
  - 4.2 POSIX thread
- 5. Introduce the LemonTea framework
  - 5.1 The book: APUE
  - 5.2 The advantage of Monk-C/C hybrid programming
  - 5.3 Future goal

1. Overview

1.1 Software Layers of Monk-C oop toolkit

Monk-C layers

The Monk-C in written by assemble and C language target for X86 and ARM architecture.
The ABI Monk-C obtained is:

IA32
X86_64
ARM32
ARM64

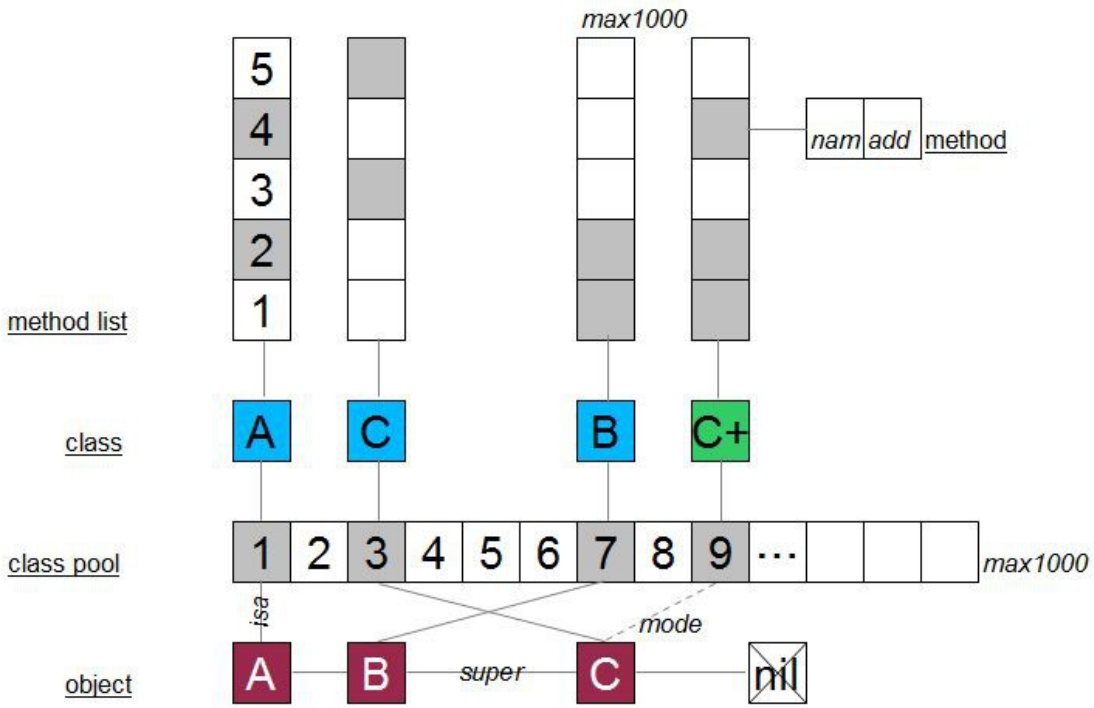
The stack control layer is a set of some short functions written by assemble to directly jump to a given address or change argument which is passed on stack.

The runtime layer is a little runtime written by C for dynamic method binding / resolving / object creating / etc

The Library layer is a library written by Monk-C which wrapped many standard C libraries this Library is called “LemonTea” because this drink is full of vitamin **C** :)

User code link the runtime “libmonkc.a” as a static library
and the static library “liblemontea.a” is optional

1.2 Data structure for oop abstract



Monk-C data structure

Monk-C is written by many C structures. As the figure above
it have object / class pool / class / method list / method five important structures to modeling the oop world.

A object A is a simple struct with **isa** pointer and **super** pointer as well as the **mode** pointer object A is linked with class A by **isa** pointer.

The class A itself is also a simple struct with **method_list** array. All the classes is put into the class pool array which have a max classes number limitation.

Inherit tree is a link list from object A itself to the root object: C (which super is nil)
when we new a object A. the runtime system actually create many structs (A, B, C)
until reach root object.
if we new(A) we get: A->B->C in memory
if we new(B) we get: B->C in memory

Method list is a simple C array filled with methods. The method is also a struct, have the **address** and **name** fields.

Attention:

- The classes in class pool have no father / child relationship. the inherit tree is dynamic. A object C can be the class of C or the class of C+ this is decided by its **mode**
- Both the class pool and method list of each class have a limitation. You should set the **MAX_CLASS_NUM** and **MAX_METHOD_NUM** macros to fit your project scale.

1.3 The power of Hash algorithm

In the section 1.2 we have see the structure of Monk-C. You may have many questions about how these C structures connected. How they find each other ?

Monk-C heavy used hash algorithm to identify and search classes / methods
in the 2d table of class and method. If you did not know it. → wiki

the hash value is actually a long binary numbers represent in hex like this:
0x0123456789abcdefgh
which is calculated from a original more longer binary number

the string in C is also a binary number inside the computer. So we can calculate the "unique" hash value from any strings. For example: method names.

Example:

hash("amethodToStartThread") → **0x0123456789abcdefgh**

hash("amethodToStopThread") → **0x9876543210zxcvbnml**

Thinking about a big array. Which contains many method addresses. We can use the hash value as index to get address. This array will be perfect for implement something dynamic because you can storage and get something by name. Just like a little database.

As the hash value is vary large. There is no example directly use the value as index. We use the M**odulo Operation** to convert the large value to a small one but keep the value unique to the name.

Example:

0x0123456789abcdefgh % 1000 = 205

method_list[205] = method_address

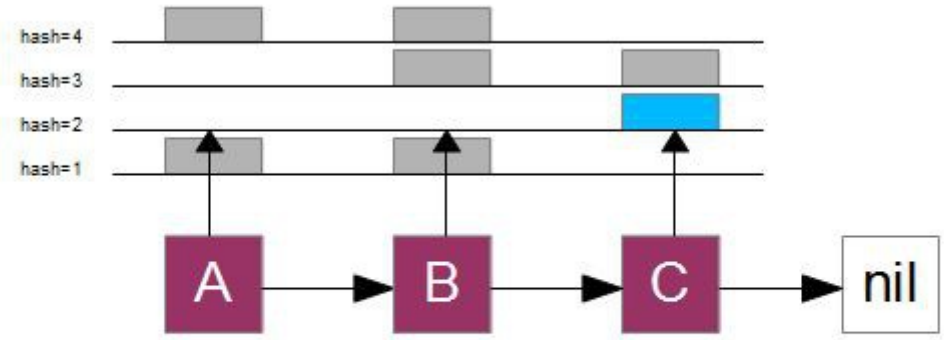
method_list[hash("methodName") % 1000] → *fetch address by name*

Both the "class pool" and the "method list" of Monk-C use the algorithm above.
The time complexity is T(n) = O(1).
And we cost **TotalClassNumber** x **MethodsPerClass** x **sizeof(pointer)** addtion bytes to which cost by pure C programes. (example: 1000 x 1000 x 4byte) 4MByte

We spend RAM space to get all the benefit of OOP with little latency (T(n) = O(1))

1.4 Method resolve

Use the algorithms in 1.3 we can resolve a method address by its name.
But that is not enough. As almost all OOP language have an important feature called: **Inherit.** Monk-C use a special method resolve logic to implement inherit feature of oop.



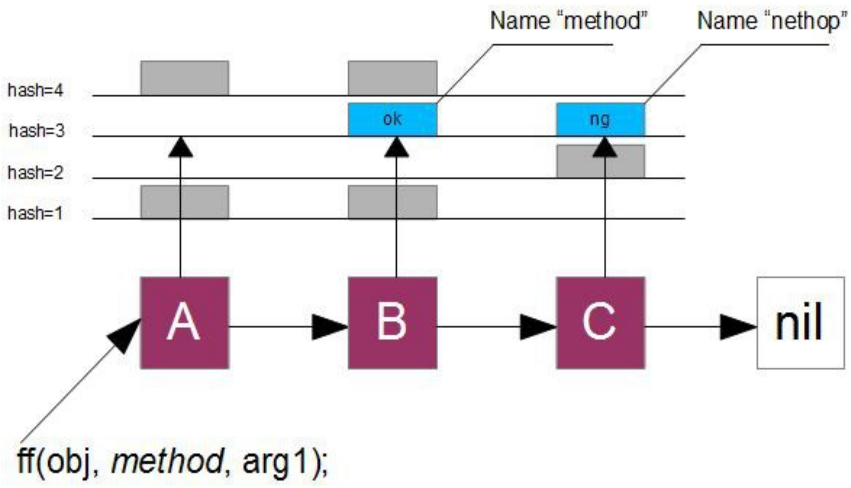
Method Resolve Logic (no conflict)

As the figure above we have an object A. A connect to B and C and nil.
All of A, B, C have its own method list.
If we call a method named “amethod” on A. the hash value will be calculated:
hash(“amethod”) % 1000 = 2

So we check the method list of A first. A have no method saved in index **2,** we go to check the next: B. B also have no method saved in index **2,** we go to C: C have a method saved at the index **2**. so call A “amethod” will cause the method saved in method list of C be called. And actually the method process on data of C.

Hash Collision:
Because it is possible two string have the same hash value. How to solve the collision effective is very important for Monk-C.
Cause the number of public methods of a class should not be so large for user to use it. Monk-C set the size of method_list 4000 and can be redefined.
All the method name hash value in a class **should be unique.** Monk-C runtime will check it at method binding step. If there is collision. Monk-C runtime request you to change the name of the last binding method and kill the process. If an object is safely created. There is no collision. Cause the size of method_list is large enough and the BKDR Hash Function is good enough. So it is safe to write a class with 100 methods without collision.

Method Resolve Logic (have conflict)



Although every class itself have "non-conflict" method list. It is possible to have a same hash value from different method in different classes. Such as the example above: "method" and "nethop" have the same hash value: 3. we can not figure out the user means to call either the "method" of B or the "nethop" of C.

Monk-C allow different class have the same name method. This is the "override" concept of OOP. Child class can implement a method use the same name with its father. Another identifier is needed to help distinguish methods when hash value is same.

The core reason of collision is the similarity of two strings. We can avoid collision by change the method name to a new more longer or shorter string. Command from user is string format. User use different string to call different method. So the prime problem is to distinguish strings. Monk-C use the origin method name string plus hash value to identify a method.

String Compare is a slow algorithm. But most of the time collision is very less. (**10 classes in inherit chain** x **each have 100 public methods = 1000**) this is the same methods number as in one class with no collision.

1.1 Software Layers of Monk-C oop toolkit
1.2 Data structure for oop abstract
1.3 Method resolve
1.4 Mode for objects
1.5 Reference counting for memory management
1.6 Floating point data
2. Syntax
2.1 Syntax for class define
2.2 Syntax for class implement

2.3 Syntax for protocol

2.4 Syntax for mode

2.5 Syntax for reference counting

2.6 Syntax for floating point data

3. Practice and Debug

3.1 The Makefile

3.2 Introduce of log system

3.3 Traps of Monk-C

3.4 Pay attention to memory management

4. Concurrency

4.1 Compare and swap algorithm for lock free implementation

4.2 POSIX thread

5. Introduce the LemonTea framework

5.1 The book: APUE

5.2 The advantage of Monk-C/C hybrid programming

5.3 Future goal