Code & Coffee #22

Low-latency Java: breaking the boundaries

Ruslan Shevchenko
<ruslan@shevchenko.kiev.ia>
@rssh1

Overview

- What is low-latency and when it needed.
- LL-specific programming techniques:
  - Concurrent flows
  - Memory
    - Layout issues,
    - GC
    - Unmanaged memory access.
  - JNI

HPC != HPC

throughput

efficiency

speed

//low latency

compromise

low-latency java:

- Ineffective
  - hardware is underused.
- Non-scalable
  - operational range is strictly defined.
- Uncomfortable
  - api reflect low-level details.

// NEEDED: SOFT REALTIME

soft realtime

- trading (HTF)

// NYSE trading floor.
// was closed for humans in 20

- video/audio processing:

- orchestration infrastructure:

softrealtime: why JVM (?)

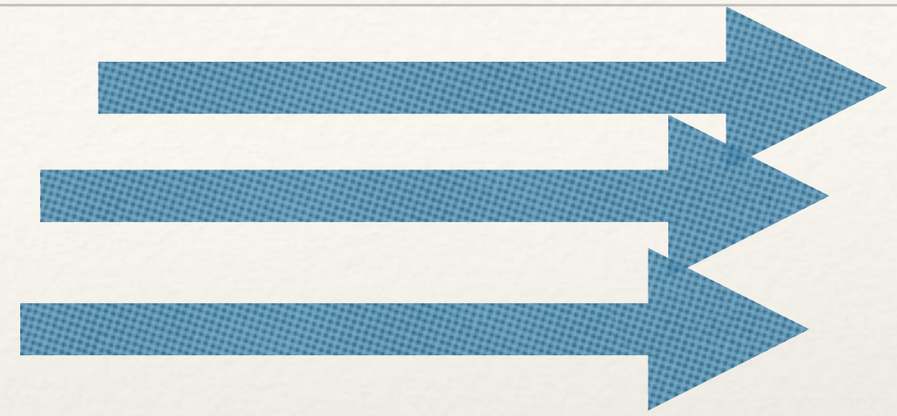
- Balance (99 % - ‘Usual code’, 1% - soft realtime)

- crossing JVM boundaries is expensive.
- don’t do this. [if you can]

Softrealtime & JVM: Programming techniques.

- Don't guess, know.
- Measure
  - profiling [not alw. practical];
  - sampling   (jvm option: -hprof) [jvisualvm, jconsole, flight recorder]
  - log safepoint / gc pauses
  - Experiments
    - benchmarks: http://openjdk.java.net/projects/code-tools/jmh/
    - critical patch: N2N flow.
- Know details, how code is executed.

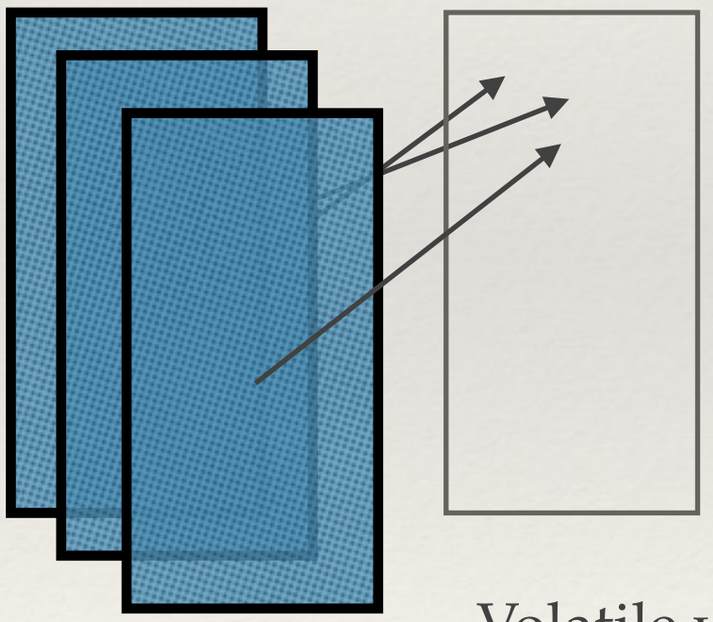
Programming techniques: Concurrent models



- Concurrent flows:
  - minimise context switches:
    - don't switch [number of flows < number of all processor cores].
      - Java: main-thread + service-thread + listener-thread; [user threads: cores-3]
- pin thread to core [thread affinity]
  - JNI [JNA], exists OSS libraries [OpenHFT]

Programming techniques: data exchange

- Concurrent flows:
  - data exchange between threads:
    - each thread have own ‘local’ memory state.



// state can be unsynchronised

// local access is match faster than volatile

L1 cache: 0.3ns
L2 cache: 1ns
L3 cache: 10ns
RAM: 120ns

Volatile write = RAM write + invalidate cache lines.

- queue between threads is faster then shared access.
- locks are evil.

Programming techniques: memory issues

- GC
  - young generation (relative short)
  - full.
    - [Stop-the-world], pause ~ heap size
- Contention
  - flushing cache lines is expensive

Soft-realtime: memory

- Object: what is java object for the JVM ?

```
class Order
{
  long    id
  int     timestamp;
  String  symbol;
  long    price;
  int     quantity;
  Side    side;
  boolean limit;
  Party   party;
  Venue   venue;
}

1002,t,”APPL”,p,q,Bu
y/Sell,  Owner,
Venue.
```

Soft-realtime: memory

- Object: what is java object for the JVM ?

```
class Order
{
   long     id;
   int      timestamp;
   String   symbol;
   long     price;
   int      quantity;
   Side     side;
   boolean  limit;
   Party    party;
   Venue    venue;
}

t,”APPL”,p,q,Buy/
Sell,  Owner, Venue.
```

```
header [64]:
class  [32|64]
id     [64]:
timestamp  32:
symbol  [32-64];
price   [64]
quantity[32]
side    [32|64]
limit   [32]
party   [32|64]
venue   [32|64]
… padding  [0-63]
```

```
header,class:128
data:  64 “APPL”
```

```
header,class: 128
 ……..
```

```
header,class:128
 ……..
```

pointer (direct or compressed)

direct = address in memory

Soft-realtime: memory

- Object: what JMV do ?

```
Order order = new Order()
```
// alloc 80 bytes from heap
// order in a local variables (GC root)
//can force GC is memory reached threshold.

```
int t = order.getTimestamp();
```
// memory read

```
Party p = order.getParty();
```
// memory read

```
order.setParty(p1);
```
// memory write +
//mark object to be rechecked in next GC cycle

Soft-realtime: memory

- How ‘idiomatic’ Java code will behave ?

```
List<Order>  send = new ArrayList<>();
while(!buffer.isEmpty()) {
    Order order = buffer.readOrder();
    int fullPrice = order.price*order.quantity;
    if (order.getParty().getLimit()+fullPrice > maxLimit) {
        client.rejected(order,”party-limit”)
    }else{
        engine.process(order);
        send.add(order);
    }
}
```

Soft-realtime: memory

- How ‘idiomatic’ Java code will behave ?

```
List<Order>  send = new ArrayList<>();
while(!buffer.isEmpty()) {
    Order order = buffer.readOrder();
    int fullPr                                          tity;
    if (order.                                          ice > maxLimit) {
        client.rejected(order,”party-limit”)
    }else{
        engine.process(order);
        send.add(order);
    }
}
```

- order will be created

- added to long-lived collections

will cause ‘StopTheWorld’

Soft-realtime: memory

- V1: Die young or live forever.

```
LongSet  send = LongSet.create();
while(!buffer.isEmpty()) {
    Order order = buffer.readOrder();
    int fullPr                                    tity;
    if (order.                                    ice > maxLimit) {
        client.rejected(order,"party-limit")
    }else{
        engine.process(order);
        send.add(order.getId());
    }
}
```

- order will be created

- collected in young generation

Soft-realtime: memory

- V2: Preallocate all.

- order will be created

```
LongSet  send = LongSet.create();
Order order = new Order();
while(!buffer.isEmpty()) {
   buffer.fillOrder(order);
   int fullPrice = order.price*order.quantity;
   if (order.getParty().getLimit()+fullPrice > maxLimit) {
       client.rejected(order,”party-limit”)
   }else{
       engine.process(order);
       send.add(order.getId());
   }
}
```

- one ‘static’ object per thread
- no allocations.

Soft-realtime: memory

- V3: Access to serialised data instead object

```
class OrderAccess
{
  static void getId(Buffer b,int offset)
     {  return b.getLong(offset+0) }

  static void putId(Buffer b, int offset, long id)
     {  b.putLong(offset+0,id) }

  static int getTimestamp(Buffer b, int offset)
     {  return b.getInt(offset+8); }
  ……..

}
```

Soft-realtime: memory

- V3: Access to serialised data instead object

```
LongSet  send = LongSet.create();
for(int offset=0; !buffer.atEnd(offset);
                                offset+=OrderAccess.SIZE) {
    long price = OrderAccess.getPrice(buffer,offset);
    int  quantity = OrderAccess.getQuantity(buffer,offset);
    int fullPrice = price * quantity;
    long partyLimit = OrderAccess.getPartyLimit(buffer,offset
    if (partyLimit > maxLimit) {
        client.rejected(buffer,offset,”party-limit”)
    }else{
        engine.process(buffer,offset);
        send.add(order.getId());
    }
}
```

- no temporary objects
- no allocations
- slightly unusual API

Soft-realtime: memory

- V4: Offheap memory access via unsafe

sun.misc.Unsafe

- internal low-level sun API
- implemented by JIT intrinsic
- JVM crash on error
- widely used but not recommer
- will be deprecated in Java-9
  - (removed in Java 10)
  - (we will discuss alternatives)

```
class Unsafe
{
  byte getByte(long addr)
  void setByte(long addr, byte value)
  ….
  byte getByte(Object value, long offs)
  ….
  //same  for primitive types.
  ….
  // access to Object internals.
}
```

Soft-realtime: memory

- V4: Offheap memory access via unsafe

```
class OrderAccess
{
   static void getId(long address)
      {  return unsafe.getLong(address) }

   static void putId(long address, long id)
      {  unsafe.putLong(offset+0,id) }

   static int getTimestamp(Buffer b, int offset)
      {  return b.getInt(offset+8); }
   ……..

}
```

Soft-realtime: memory

- V4: Offheap memory access via unsafe

```
LongSet  send = LongSet.create();
for(long address=buffer.begin; address!=buffer.end();
              address=advance(address,OrderAccess.SIZE) ) {
    long price = OrderAccess.getPrice(address);
    int  quantity = OrderAccess.getQuantity(address);
    int fullPrice = price * quantity;
    long partyLimit = OrderAccess.getPartyLimit(address)
    if (partyLimit > maxLimit) {
        client.rejected(address,”party-limit”)
    }else{
        engine.process(address);
        send.add(order.getId());
    }
}
```

- no allocations
- any memory
- subtle bugs are possible.

// better wrap by more hight-level interfaces (buffer, etc)

Soft-realtime: memory

- V3, V4 Problem: Verbose API

  - bytecode transformation from more convenient API

  - Metaprogramming within hight-level JVM languages.
    http://scala-miniboxing.org/ildl/
    https://github.com/densh/scala-offheap

- Integration with unmanaged languages

- JVM with time-limited GC pauses. [Azul Systems]

Memory access API in next JVM versions

- VarHandlers (like methodHandlers)
    - inject custom logic for variable access
    - JEP 193 http://openjdk.java.net/jeps/193
- Value objects.
    - can be located on stack
    - JEP 169. http://openjdk.java.net/jeps/169
- Own intrinsics [project Panama]
    - insert assembler instructions into JIT code.
    - http://openjdk.java.net/projects/panama/
    - JEP 191. http://openjdk.java.net/jeps/191
- ObjectLayout [Arrays 2.0]
    - better contention
    - http://objectlayout.github.io/ObjectLayout/

Low latency Java: access to native code

- JNA
  - (idea — API to build native stack call and get result)
  - pro: not require packaging native libraries.
  - cons: SLOWWWW (but one-time: ok)
- JNI
  - (idea — bridge between JNI/native code)
  - pro: faster than JNA, exists critical jni
  - cons: need packaging native library
- Shares Memory
  - (idea — use same binary address from native & jvm code)
  - pro: fast
  - cons: unsafe, unusual API

Low latency Java: JNI

- JNI
  - (idea — bridge between JNI/native code)
  - Java Object => Handle for native (maintain table)

A Sample Native Call

```
        save 64           # register window push
        add   g7,&jnienv_offset,o0  # JNIEnv* in o0
        set  0,o1         # assume null arg
        beq  i0,skip      # handle of null is null
        stw  i0,[sp+72]   # handlize 'this'
        add  sp,72,o1     # ptr to 'this' passed in o1
skip:
        std  d0,[sp+64]   # float args passed in int regs
        ldd  [sp+64],o2   # double now in o2/o3
        setlo #ret_pc,10
        sethi #ret_pc,10  # set 32-bit PC before allow GC
        stw   10,[g7+&pc] # Rely on Sparc TSO here (IA64!)
        stw   sp,[g7+&sp] # Enable GC from here on
        call  foo         # The Native Call
ret_pc: add   g7+&sp,10   # CAS does not take an offset
        cas   0,sp,[10]   # if SP still there, store 0 to disable gc
        bne   gc_in_progress
        set   0,i0        # assume null result
        beq   o0,is_null
        ldw   [o0],i0     # de-handlize return value
is_null:
        return            #
        restore           # pop sparc window
```

Handles, GC-unsafe natives

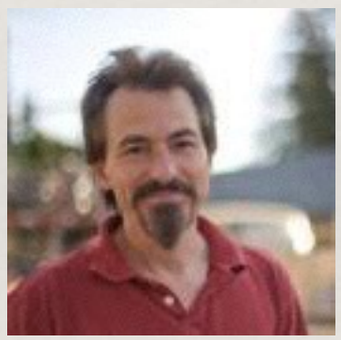
Arg-shuffle JIT vs native convention

Stack crawl GC lock

The Call

GC unlock

De-Handlize



// part of Cliff Click presentation (advise for VM writers)

Low latency Java: JNI

- JNI
  - (idea — bridge betwee
  - Java Object => Handle

```
class JNIWrapper
{
  void native eventLoop();
  void processCallback(Packet p);
}
```

- Callbacks are extremely slow

C:

```
extern Java_nativeEventLoop(JEnv jenv, JObject job) {

  while(!endOfBuffer(buff)) {
    packet p = getData(buff);
    if (checked(p)) {
      processInJavaCallback(p);
    }
  }
}
```

- slow

- p will be marshalled / unmarshalled each time

Low latency Java: JNI

/ / better use return instead callback

```
extern Java_nativeEvent(JEnv jenv, JObject job) {
  static int state = START;
  switch(state) {
      case START:
         state = LOOP;
         while(! endOfBuffer()) {
      case LOOP:
            Packet p = readData();
            if (checked(p)) {
               return (long)p;
            }
         }
  }
  return -1;
}
```

```
class JNIWrapper
{
  void native event();
  void process(long packetAddr
}

  …

for(;;) {
  p=event();
  if (p==-1L) break;
  process(p);
}
```

Low latency Java: JNI

- Critical JNI
  - not in official documentation
  - substituted instead JNI after JIT warm-up.
  - only primitive types; static calls.
  - must not be long-running. (GC is disabled)

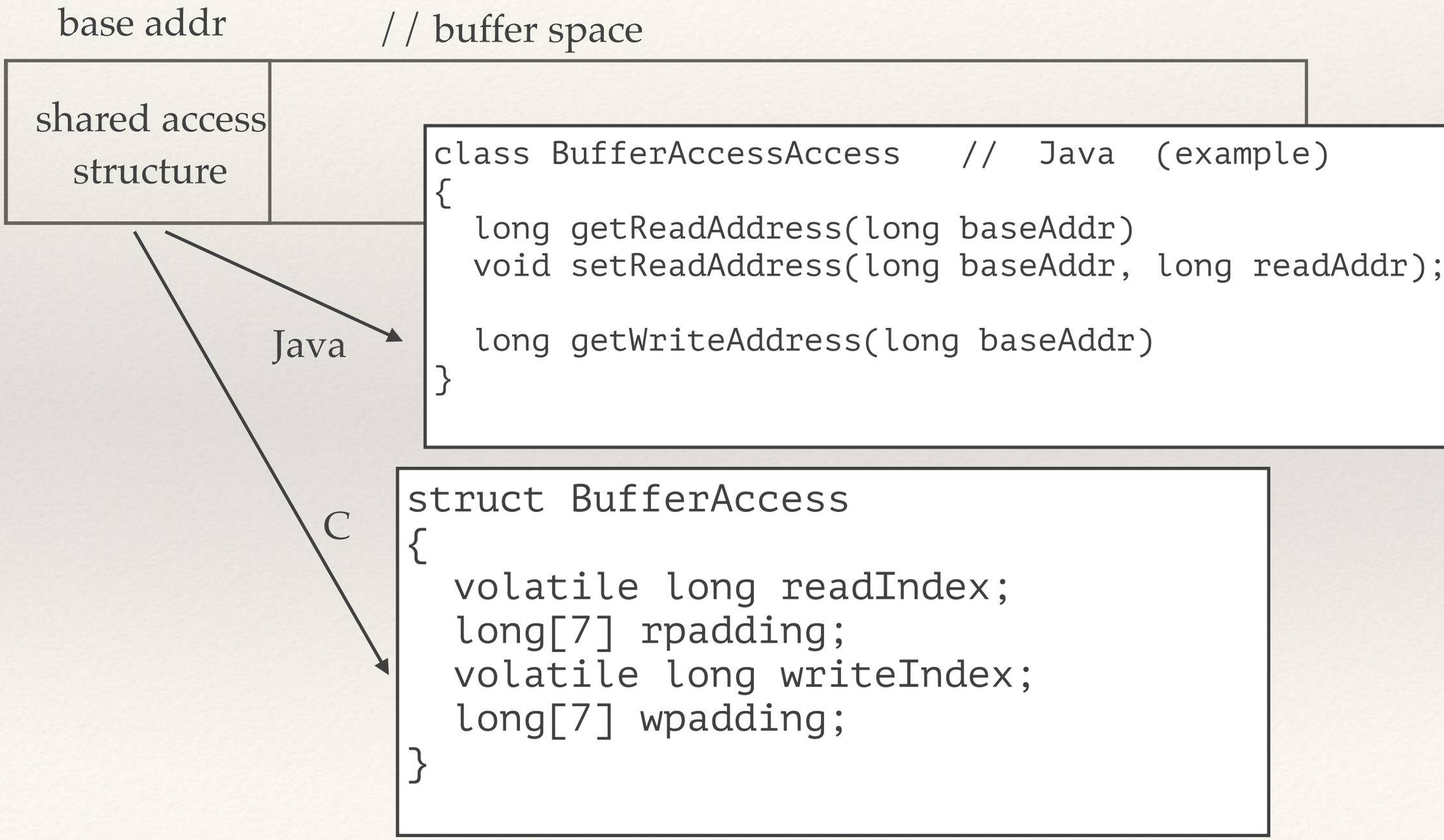
```
extern long Java_nativeEvent(JEnv jenv, JObject job)
->
      extern long JavaCritical_nativeEvent()
```

// want add timeout to satisfy ‘last’ condition

Low latency java: shared memory interface

- idea:
  - spawn process or native thread.
  - mmap file with given name to memory on ‘native side’.
  - mmap file with given name to memory on ‘java’ side.
  - have shared memory persist to the same file in java & native code.

Low latency Java/Native code data exchange



Low latency Java/fw

- Low-latency on JVM is
  - possible
  - sometimes non-trivial
  - special techniques exists, in future will be better.

Thanks for attention.

- Questions ?

// ruslan@shevchenko.kiev.ua
// @rssh1
// https://github.com/rssh