多线程架构的几个解决方案

我做游戏开发以来，历经了《华夏》《远征》等多个产品，之前的这几个游戏的服务器设计总体都是单线程结构。特别是早期的《华夏》，甚至是单服务器架构，连服务器集群都没有。

实际运营带来的反馈就是，由于单线程结构编码简单，所以总体开发效率和程序稳定性较高。

但是由此带来的问题就是，一是负载不够均衡，二是响应延时较大。负载上只能通过多开场景服来分流。但常常服务器压力大时反而是一大群人集中在一个场景进行大型活动的时候，此时分流效果不理想，也没有发挥多核CPU的优势。

后经过了解，韩国的动作游戏多是采用多线程结构，大体一个角色一个线程。最初我对此架构呲之以鼻，后12年后各种采用Erlang结构的页游兴起，以及《远征》重写通讯层代码时，积累了一些多线程编码的经验，我发现采用多线程架构并非不可行，主要是结构上需要把一些多线程同步的方案界定清楚。

1. **线程间通讯的方式**

根据负载的切分，将系统的实现分成多个线程后，由于游戏逻辑不可能完全独立，肯定涉及多个线程之间互相通讯的问题。常见的线程间通讯方式有：

1. *.异步消息*

发送消息是跨线程通讯最简单高效的方法。如果要设计一个多线程架构的服务器，消息模式一定要被重点使用。 因为该方法编码时调用简单，不需要考虑多线程同步，

唯一有一大硬伤，就是无法返回调用结果。对于编程来说，只能通知做什么事情，不能

用来get什么东西。就有一半函数功能无法实现。

1. *. RPC回调*

由于消息是异步处理的，不能及时返回结果，所以应用时不能用来实现get类函数

另外也不能实现对处理有顺序要求的逻辑。例如某个逻辑必须先创建完一个对象，再进

进行下一段逻辑。针对这个问题，《远征》曾经设计过一个来往于服务器集群之间的RPC

调用，内部是基于消息的封装，即

RPC\_Call( zone\_server, method\_zone\_server\_xxxxxx, id,param, callback\_social\_server\_xxxx)

一个典型的使用场景是，社会服通过RPC消息调用场景服method\_zone\_server\_xxxxxx

函数，调用成功后再通过消息回调社会服callback\_social\_server\_xxxx函数，双方都将参

数打包在调用现场里。此种结构通过回调能比较好的解决顺序问题，缺点就是代码要分

成多个函数，增加了一定编码复杂度。这种只适合大块的逻辑，对于get类函数就不适

合，会将逻辑分得过于细碎。

1. . *函数调用*

对于get类函数，效率最高的就是通过加锁，或者线程安全的无锁编码，然后直接

调用。由于get类多为获取对象属性，所以可以底层实现一个线程安全的属性集，上层

代码直接创建一个线程安全属性集，然后跨线程直接存取数据，而并不需要了解线程安

全编码的细节。Erlang我记得也是类似消息+record的结构，record就是可以跨线程访

问的数据。

总体来说直接函数调用只能适用少部分简单的多线程安全的函数。对代码编写者要

求较高。

1. .*阻塞式消息*

阻塞式消息的流程是假设A要调用B的某个函数并返回结果。 程序内部的实现是

A向B发送一个消息，并自身阻塞休眠，B处理该消息后，将返回结果丢到调用现

场里，A收到结果，解除休眠，继续执行下一步逻辑。

阻塞式消息对编程者非常友好，可以和单线程一样编写逻辑，但不能调用过于频繁，

因为阻塞的效率并不高。

另外逻辑上应避免阻塞调用的嵌套死锁。例如A调用B的某个的某个函数，而B

的这个函数又要阻塞调用A的某个函数才能返回。而此时A已经是阻塞状态无法

处理消息，由此出现死锁。所以阻塞调用的函数，最好是封闭的函数。 即其中

不应该再包括其他阻塞函数的调用。

1. **对象生命周期**

我在写《远征》的通讯层代码时，有很长时间通讯层总是各种崩溃。基本都是多线程中对象生命周期的管理问题。 直到一年之后重构第三遍，把所有的相关对象采用share\_ptr的方案来管理，问题才得以解决。

1. *.多线程之间对象标识*

如果有2个不同线程A和B，如果要互相对象，此时有2种方式：

1. . **句柄 + API**  : 即每个线程可被外界调用的函数都封装成一个API

b). **智能指针 + 接口函数** : 即每个线程的对象暴露出智能指针，拿指针调用接口函数

句柄其实内部还是要转化成对象指针，所以2者本质是一样的。

1. *.多线程对象生命周期安全*

最典型的对象生命周期安全的问题：

以下代码示例A线程调用B线程的func\_1()函数

Call\_Fun\_1( handle\_B)

{

B \* pB = getHandle(handle\_B);

If ( pB == 0) return; **// 此时B线程可能已经Release,导致pB虽然不是0但已销毁**

pB->func\_1();  
 }

因为2个线程的调用代码，做不到在一个原子操作中完成，所以只要其他线程保存了指

针，该指针是否有效，完全得不到保证，代码就一定会崩溃。

1. *.智能指针的解决方案*

要解决上面描述的问题，目前只能想到智能指针方案。 即对象的Release不能由某个

线程单独负责（他需要管其他人有没有引用）。每次引用对象都增加一个引用计数，

引用计数到0时才真正Release对象。该方案非常清晰的解决了对象生命周期的问题，

但也有点小问题：

3.1). **智能指针的赋值本身不是线程安全的**

由于智能指针的实现需要2个变量，一个Count一个Ptr,导致不能在一个原子

操作中完成。所以严格意义上在赋值时可能出现安全问题。

3.2).**释放对象时的线程问题**

如上所述，对象的Release不再由创建他的线程管理，那么可能对象在A线程

创建，B线程最后一个引用，就会在B线程真正Release。对于简单的对象来讲

可能随便在哪个线程释放都可以。但有些对象Release时要做一堆操作，而这

堆操作不是线程安全的，此时会出现各种问题。

1. *.智能指针完整方案*

综上所述，智能指针最重要的是解决线程间拿到一个坏指针的问题。并应该把释放

逻辑也搞乱。所以较为完整的方案应该是：

1. 每个对象增加一个destroy(),用来销毁对象，但不delete this. Destroy()只在创建的

线程调用

1. Destry()之后的指针，依然有可能被其他线程引用，此时类接口暴露的方法都要判

断一下，如果is\_destroy则直接返回。

3.当所有线程引用计数为0时，此时才Release对象，调用delete this.

4.虽然智能指针赋值并不是原子的，但貌似只要保证获取智能指针时是安全的就问

题不大，因为Release时引用计数已为0，不太可能再取这个指针。另外可能用

智能指针的指针来保证原子操作。 这个问题还没想太明白，但貌似问题不大，

另案再表.

1. **如果划分线程和逻辑**

由于跨线程调用方法的诸多限制。为了提高一个多线程架构的性能，需要尽可能的使用异步消息模式，需要避免阻塞式消息的死锁。线程和逻辑划分不好，则阻塞死锁多。线程划分好，则基本是异步消息和get类高性能的调用，整个系统才能充分的发挥多线程架构的性能优势，反之则到处都是阻塞死锁，反而消息更低。

如何划分线程和逻辑，最好是人物，怪物，场景， 这些是简单的MODEL，他们只负责自身逻辑，一般不调别人。战场，LUA，这些是Controller，他们会调多个别人，阻塞比较多，但不会出现死锁。 如果MODEL中间互相调，Controller又调model,model又调Controller,则经常出现死锁互等逻辑。

比较好的设计应该是多个MODEL + 2个controller

MODEL : 人物，怪物 只负责自身逻辑，基本不外调

Controller\_1 (低延迟的) : Buff 技能 广播

Controller\_2 (高延迟的) : AI 战场逻辑

1. **创建销毁的异步处理**

异步编程时经常会出现这样的逻辑。 创建一个人物，等人物创建完后，再加入到战场

并发送XX消息。

1. 事件
2. 回调
3. 阻塞
4. **如何使用锁**

1.自旋锁

2.读写锁

3.无锁

4.尽可能减少锁的时间

5.锁的效率和自旋

1. **如何使用原子操作**