游戏服务端究竟解决了什么问题？ - 文章 - 伯乐在线

1.写在前面

既然是游戏服务端程序员，那博客里至少还是得有一篇跟游戏服务端有关的文章，今天文章主题就关于游戏服务端。

写这篇博客之前也挺纠结的，一方面是因为游戏服务端其实不论架构上还是具体一些逻辑模块的构建，都属于非常成熟的技术，举个简单的例子，像端游的多zone/scene/game进程+单全局进程架构，网上随便一搜能搜出来几十篇内容差不多的。另一方面是因为中国特色MMO基本上把服务端程序员整成了业务逻辑狗，很多明星团队的业务狗基本上从入职第一天开始就成天写lua、写python，纯写lua/python，你是完全无法辨别一个程序员的vision强弱区别的，结果论资排辈导致vision弱的上去了。（也许vision强的出去创业了？）你就会发现，游戏服务端的话语权到底是被谁占据了。

在我看来，游戏服务端程序员容易陷入两个误区：

第一，游戏服务端实际上要解决的并不是性能问题。一方面，即使是千人同屏的端游（姑且不论这千人同屏是不是一个中国特色的伪需求，反正我是没法将千人同屏跟游戏乐趣联系在一起的），其服务端如果进程划分得当，一个场景进程也至多只有千级别entity的压力，性能问题退化为了逻辑狗的业务素养问题。另一方面，现在端游MOBA和手游时代，开房间式场景同步已经成为主流，各种逻辑狗进化来的资深人士不需要也没必要将性能挂在嘴边了。

第二，大部分游戏服务端所谓框架的定位有误。服务端框架的设计有好有坏，判断一个设计好不好没有普适统一的标准，但是判断一个设计烂不烂一定是存在一个标准线的。简单列举几种烂设计：

烂设计基础版本。帮你定义好框架中的几种角色，你要么全盘接受，要么全不接受，不存在中间状态。但是，提供一种简单的通信机制，以及外部与框架通信的clientLib。或者能让你定制开发其中一种角色，可以写外部driver。这样，虽然架构丑一点，至少还能提供一定程度的扩展性。

烂设计进阶版本。除了满足基础版本的定义之外，还具有一些额外的烂特点：框架中的角色定义的特别二逼，举个例子，基础版本的烂设计在角色定义上可能只是大概区分了Db代理进程、Gate进程、逻辑进程，但是进阶版本会对逻辑进程进行区分，定义了不同的逻辑进程角色。这意味着什么？意味着我想写一个简单的单逻辑进程游戏是没办法用这个框架的，因为框架默认就集成进来了一堆莫名其妙的东西。更有甚者，我想要添加一种角色，是需要动手去改框架的。

说实话，正是由于这类设计的存在，我在看到类似于“游戏服务端技术含量不高”这类论断的时候，总感觉辩无可辩，因为就这两种设计而言，我甚至除了代码逻辑复杂度之外看不到跟本科毕设级别的游戏服务器有什么区别。

不知道算是不幸还是幸运，前段时间亲眼目睹了上述提到的某种设计的从无到有的过程。当然，今天写

此文的目的不是为了将这种设计批判一番，每种设计的诞生都是与各种因素相关的，我们不能站在上帝视角去评判这个过程。今天写此文，是希望对自己这整整一年半的游戏服务端编码历程中的一些所思所惑做个整理，希望能带各位看官从另一个思路看游戏服务端。

2.游戏服务端究竟解决了什么问题？

从定义问题开始，简单直接地说，一套游戏服务端开发框架应该具有下面两种能力：

- 定义了client到server、server到client、server到server的消息pipeline。
- 描述了游戏世界状态的维护方式。

下面就从这两点来展开这篇文章。

3 消息pipeline

3.1 经典消息pipeline

3.1.1 场景同步

当讨论到游戏服务端的时候，我们首先想到的会是什么？要回答这个问题，我们需要从游戏服务端的需求起源说起。

定义问题

游戏对服务端的需求起源应该有两个：

- 第一种是单机游戏联网版，实现为主客机模式的话，主机部分可以看做服务端。
- 第二种是所有mmo的雏形mud，跟webserver比较类似，一个host服务多clients，表现为cs架构。

第一种需求长盛不衰，一方面是console游戏特别适合这一套，另一方面是最近几年手游起来了，碎片化的PVE玩法+开房间式同步PVP玩法也得到验证，毕竟MMO手游再怎么火也不可能改变手游时间碎片化的事实的，最近的皇家冲突也证明，手游不会再重走端游老路了。

第二种需求就不用说了，网上大把例子可以参考。最典型的是假设有这样一块野地，上面很多玩家和怪，逻辑都在服务端驱动，好了，这类需求没其他额外的描述了。

但是，解决方案毕竟是不断发展的，即使速度很慢。

说不断发展是特指针对第一种需求的解决方案，发展原因就是国情，外挂太多。像war3这种都还是纯正的主客机，但是后来对战平台出现、发展，逐渐过渡成了cs架构。真正的主机 其实是建在服务器的，这样其实服务器这边也维护了房间状态。后来的一系列ARPG端游也都是这个趋势，服务端越来越重，逐渐变得与第二种模式没什么区别。 同理如现在的各种ARPG手游。

说发展速度很慢特指针对第二种需求的解决方案，慢的原因也比较有意思，那就是wow成了不可逾越的鸿沟。bigworld在wow用之前名不见经传，wow用了之后国内厂商也跟进。发展了这么多年，现在的无缝世

界服务端跟当年的无缝世界服务端并无二致。发展慢的原因就观察来说可能需求本身就不是特别明确，MMO核心用户是重社交的，无缝世界核心用户是重体验的。前者跑去玩了天龙八部和倩女不干了，说这俩既轻松又妹子多；后者玩了console游戏也不干了，搞了半天MMO无缝世界是让我更好地刷刷刷的。所以仔细想想，这么多年了，能数得上的无缝世界游戏除了天下就是剑网，收入跟重社交的那几款完全不在一个量级。

两种需求起源，最终其实导向了同一种业务需求。传统MMO架构（就是之前说的天龙、倩女类架构），一个进程维护多个场景，每个场景里多个玩家，额外的中心进程负责帮玩家从一个场景/进程切到另一个场景/进程。bigworld架构，如果剥离开其围绕切进程所做的一些外围设施，核心工作流程基本就能用这一段话描述。

抽象一下问题，那我们谈到游戏服务端首先想到的就应该是多玩家对同一场景的view同步，也就是场景服务。

本节不会讨论帧同步或是状态同步这种比较上层的问题，我们将重点放在数据流上。

如何实现场景同步？

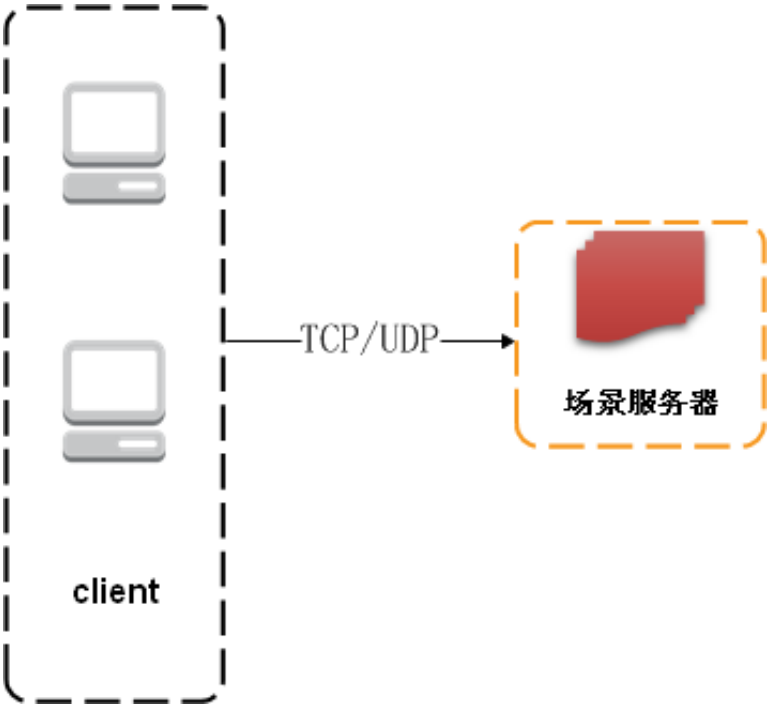
首先，我们看手边工具，socket。

之所以不提TCP或UDP是因为要不要用UDP自己实现一套TCP是另一个待撕话题，这篇文章不做讨论。因此，我们假设，后续的实现是建立在对底层协议一无所知的前提之上的，这样设计的时候只要适配各种协议，到时候就能按需切换。

socket大家都很熟悉，优点就是各操作系统上抽象统一。

因此，之前的问题可以规约为：如何用socket实现场景同步？

拓扑结构是这样的（之后的所有图片连接箭头的意思表示箭头指向的对于箭头起源的来说是静态的）：



场景同步有两个需求：

- low latency
- rich interaction

要做到前者，最理想的情况就是由游戏程序员把控消息流的整套pipeline，换句话说，就是不借助第三方的消息库/连接库。当然，例外是你对某些第三方连接库特别熟悉，比如很多C++服务端库喜欢用的libevent，或者我在本篇文章提供的示例代码所依赖的，mono中的IO模块。

要做到后者，就需要保持场景同步逻辑的简化，也就是说，场景逻辑最好是单线程的，并且跟IO无关。其核心入口就是一个主循环，依次更新场景中的所有entity，刷新状态，并通知client。

正是由于这两个需求的存在，网络库的概念就出现了。网络库由于易于实现，概念简单，而且笼罩着“底层”光环，所以如果除去玩具性质的项目之外，网络库应该是程序员造过最多的轮子之一。

那么，网络库解决了什么问题？

抛开多项目代码复用不谈，网络库首先解决的一点就是，将传输层的协议（stream-based的TCP协议或packet-based的UDP协议）转换为应用层的消息协议（通常是packet-based）。对于业务层来说，接收到流和包的处理模型是完全不同的。对于业务逻辑狗来说，包显然是处理起来更直观的。

流转包的方法很多，最简单的可伸缩的non-trivial buffer，ringbuffer，bufferlist，不同的结构适用于不同的需求，有的方便做zero-copy，有的方便做无锁，有的纯粹图个省事。因为如果没有个具体的testcast或者benchmark，谁比谁一定好都说不准。

buffer需要提供的语义也很简单，无非就是add、remove。buffer是只服务于网络库的。

网络库要解决的第二个问题是，为应用层建立IO模型。由于之前提到过的场景服务的rich interaction的特点，poll模型可以避免大量共享状态的存在，理论上应该是最合适场景服务的。所谓poll，就是IO线程准备好数据放在消息队列中，用户线程负责轮询poll，这样，应用层的回调就是由用户线程进入的，保证模型简单。

而至于IO线程是如何准备数据的，平台不同做法不同。linux上最合适的做法是reactor，win最合适的做法就是proactor，一个例外是mono，mono跑在linux平台上的时候虽然IO库是reactor模型，但是在C#层面还是表现为proactor模型。提供统一poll语义的网络库可以隐藏这种平台差异，让应用层看起来就是统一的本线程poll，本线程回调。

网络库要解决的第三个问题是，封装具体的连接细节。cs架构中一方是client一方是server，因此连接细节在两侧是不一样的。而由于socket是全双工的，因此之前所说的IO模型对于任意一侧都是适用的。

连接细节的不同就体现在，client侧，核心需求是发起建立连接，外围需求是重连；server侧，核心需求是接受连接，外围需求是主动断开连接。而两边等到连接建立好，都可以基于这个连接构建同样的IO模型就可以了。

现在，简单介绍一种网络库实现。

- 一个连接好的socket对应一个connector。
- connector负责向上提供IO模型抽象（poll语义）。同时，其借助维护的一个connector_buffer，来

实现流转包。

- 网络库中的client部分主要组件是ClientNetwork，维护连接（与重连）与一条connector。
- 网络库中的server部分主要组件是ServerNetwork，维护接受连接（与主动断开）与N条connector。
- Network层面的协议非常简单，就是len+data。

具体代码不再在博客里贴了。请参考：Network

引入新的问题

如果类比马斯洛需求中的层次，有了网络库，我们只能算是解决了生理需求：可以联网。但是后面还有一系列的复杂问题。

最先碰到的问题就是，玩家数量增加，一个进程扛不住了。那么就需要多个进程，每个进程服务一定数量的玩家。

但是，给定任意两个玩家，他们总有可能有交互的需求。

对于交互需求，比较直观的解决方案是，让两个玩家在各自的进程中跨进程交互。但是这就成了一个分布式一致性问题——两个进程中两个玩家的状态需要保持一致。至于为什么一开始没人这样做，我只能理解为，游戏程序员的计算机科学素养中位程度应该解决不了这么复杂的问题。

因此比较流行的是一种简单一些的方案。场景交互的话，就限定两个玩家必须在同一场景（进程），比如攻击。其他交互的话，就借助第三方的协调者来做，比如公会相关的通常会走一个全局服务器等等。

这样，服务端就由之前的单场景进程变为了多场景进程+协调进程。新的问题出现了：

玩家需要与服务端保持多少条连接？

一种方法是保持O(n)条连接，既不环保，扩展性又差，可以直接pass掉。

那么就只能保持O(1)条连接，如此的话，如何确定玩家正与哪个服务端进程通信？

要解决这个问题，我们只能引入新的抽象。

3.1.2 Gate

定义问题

整理下我们的需求：

- 玩家在服务端的entity可以在不同的进程中，也可以移动到同一个进程中。
- 玩家只需要与服务端建立有限条连接，即有访问到任意服务端进程的可能性。同时，这个连接数量不会随服务端进程数量增长而线性增长。

要解决这些需求，我们需要引入一种反向代理（reverse proxy）中间件。

反向代理是服务端开发中的一种常见基础设施抽象（infrastructure abstraction），概念很简单，简单说就是内网进程不是借助这种proxy访问外部，而是被动地挂在proxy上，等外部通过这种proxy访问内

部。

更具体地说，反向代理就是这样一种server：它接受clients连接，并且会将client的上行包转发给后端具体的服务端进程。

很多年前linux刚支持epoll的时候，流行一个c10k的概念，解决c10k问题的核心就是借助性能不错的反向代理中间件。

游戏开发中，这种组件的名字也比较通用，通常叫Gate。

Gate解决了什么问题

- 首先，Gate作为server，可以接受clients的连接。这里就可以直接用我们上一节输出的网络库。同时，其可以接受服务端进程（之后简称backend）的连接，保持通信。
- 其次，Gate能够将clients的消息转发到对应的backend。与此对应的，backend可以向Gate订阅自己关注的client消息。对于场景服务来说，这里可以增加一个约束条件，那就是限制client的上行消息不会被dup，只会导到一个backend上。

仅就这两点而言，Gate已经能够解决上一节末提出的需求。做法就是client给消息加head，其中的标记可以供Gate识别，然后将消息路由到对应的backend上。比如公会相关的消息，Gate会路由到全局进程；场景相关的消息，Gate会路由到订阅该client的场景进程。同时，玩家要切场景的时候，可以由特定的backend（比如同样由全局进程负责）调度，让不同的场景进程向Gate申请修改对client场景相关消息的订阅关系，以实现将玩家的entity从场景进程A切到场景进程B。

站在比需求更高的层次来看Gate的意义的话，我们发现，现在clients不需要关注backends的细节，backends也不需要关注clients的细节，Gate成为这一pipeline中唯一的静态部分（static part）。

当然，Gate能解决的还不止这些。

我们考虑场景进程最常见的一种需求。玩家的移动在多client同步。具体的流程就是，client上来一个请求移动包，路由到场景进程后进行一些检查、处理，再推送一份数据给该玩家及附近所有玩家对应的clients。

如果按之前说的，这个backend就得推送N份一样的数据到Gate，Gate再分别转给对应的clients。

这时，就出现了对组播（multicast）的需求。

组播是一种通用的message pattern，同样也是发布订阅模型的一种实现方式。就目前的需求来说，我们只需要为client维护组的概念，而不需要做inter-backend组播。

这样，backend需要给多clients推送同样的数据时，只需要推送一份给Gate，Gate再自己dup就可以了——尽管带来的好处有限，但是还是能够一定程度降低内网流量。

那接下来就介绍一种Gate的实现。

我们目前所得出的Gate模型其实包括两个组件：

- 针对路由client消息的需求，这个组件叫Broker。Broker的定义可以参考zguide对DEALER+ROUTER pattern的介绍。Broker的工作就是将client的消息导向对应的backend。
- 针对组播backend消息的需求，这个组件叫Multicast。简单来说就是维护一个组id到clientIdList的映射。

Gate的工作流程就是，listen两个端口，一个接受外网clients连接，一个接受内网backends连接。

Gate有自己的协议，该协议基于Network的len+data协议之上构建。

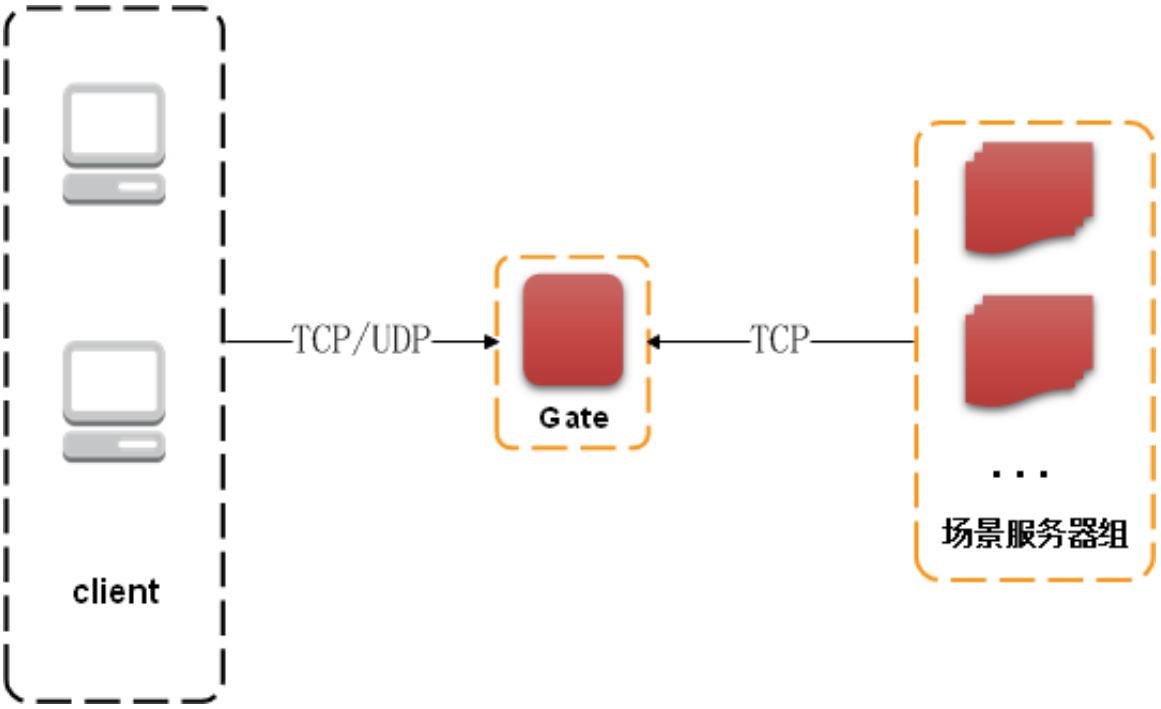
clients的协议处理组件与backends的协议处理组件不同，前者只处理部分协议（不会识别组控制相关协议，订阅协议）。

在具体的实现细节上，判断一个client消息应该路由到哪个backend，需要至少两个信息：一个是clientId，一个是key。

同一个clientId的消息有可能会路由到不同的backend上。

当然，Gate的协议设计可以自由发挥，将clientId+key组成一个routingKey也是可以的。

引入Gate之后的拓扑：



具体代码请参考：GateSharp

引入新的问题

现在我们在需求的金字塔上更上了一层。之前我们是担心玩家数量增长会导致服务端进程爆掉，现在我们已经可以随意扩容backend进程，我们还可以通过额外实现的全局协调者进程来实现Gate的多开与动态扩容。甚至，我们可以通过构建额外的中间层，来实现服务端进程负载动态伸缩，比如像bigworld那样，在场景进程与Gate之间再隔离出一层玩家agent层。

可以说，在这种方案成熟之后，程序员之间开始流行“游戏开发技术封闭”这种说法了。

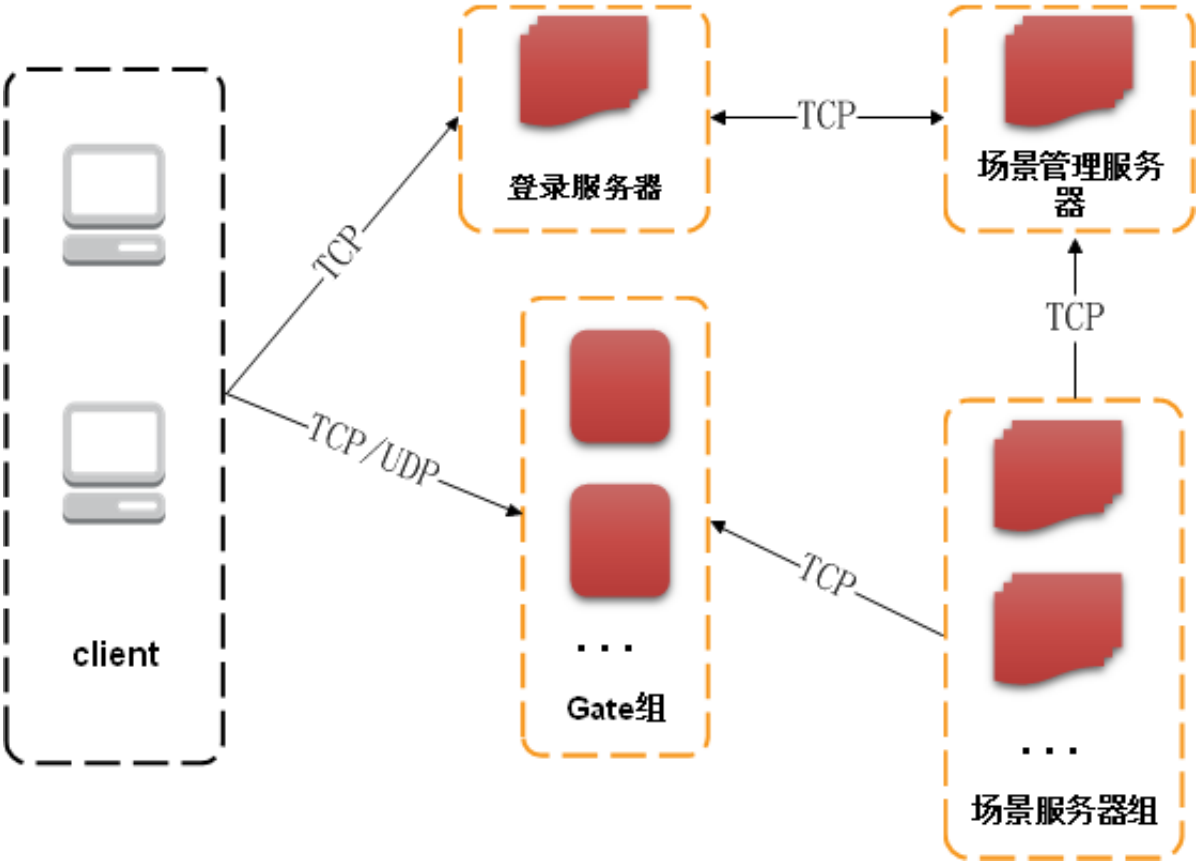
为什么？

举一个简单的例子，大概描述下现在一个游戏项目的服务端生命周期状况：

- 第一阶段，大概到目前这篇文章的进度为止，实现了场景内跑跳打。
- 第二阶段，疯狂地为场景进程增加逻辑，各种跟游戏有关的逻辑全加进来，直到这部分的代码量占到整个服务端代码量的80%以上。
- 第三阶段，有节操的程序员考虑拆分进程，当然，一开始的协调者进程一直都会存在，毕竟有些需求是场景进程无论如何都实现不了的。拆分进程的典型例子有聊天、邮件、公会等等。拆分出来的进程基本上是对场景进程代码轮廓的拷贝粘贴，删掉逻辑就开始在这之上写了。

结果就是，产出了几个玩具水平的服务器进程。要非得说是工业级或者生产环境级别的吧，也算是，毕竟bugfix的代码的体量是玩具项目比不了的。而且，为了更好地bugfix，通常会引入lua或者python，然后游戏逻辑全盘由脚本构建，这下更方便bugfix了，还是hotfix的，那开发期就更能随便写写写了，你说架构是什么东西？

至于具体拓扑，可以对着下图脑补一下，增加N个节点，N个节点之间互相连接。



玩具水平的项目再修修补补，也永远不会变成工艺品。

skynet别的不说，至少实现了一套轻量级的actor model，做服务分离更自然，服务间的拓扑一目了然，连接拓扑更是优雅。网易的mobile_server，说实话我真的看不出跟bigworld早期版本有什么区别，连接拓扑一塌糊涂，完全没有服务的概念，手游时代了强推这种架构，即使成了几款过亿流水又怎样？

大网易的游戏开发应届生招聘要求精通分布式系统设计，就mobile_server写出来的玩具也好意思说是“分布式系统”？

很多游戏服务端程序员，在游戏服务端开发生涯结束之前，其接触的，或者能接受的设计基本到此为止。如果是纯MMO手游，这样做没什么，毕竟十几年都这样过来了，开发成本更重要。更搞笑的是社交游戏、异步战斗的卡牌游戏也用mobile_server，真搞不明白怎么想的。

大部分游戏服务端实现中，服务器进程是原子单位。进程与进程之间的消息流建立的成本很低，结果就是服务端中很多进程互相之间形成了O(n^2)的连接数量。

这样的话会有什么问题？

一方面，连接拓扑关系很复杂。一种治标不治本的方法是抬高添加新进程的成本，比如如非必要上面不会允许你增加额外进程，这样更深度的解耦合就成了幻想。

另一方面，游戏服务端的应用层与连接层难以分离。举个例子，在这种设计思路下，两个进程有没有连接是一种不确定态，设计的时候觉得没有，结果某个需求来了，不建立连接就很难实现。这样对于应用层来说，就需要提供连接的概念，某个进程必须先跟其他进程连接建立成功了，然后才能调用其他进程提供的服务。而实际上，更优雅的设计是应用层完全不关注连接细节，只需要知道其他的进程提供了服务，自己就能获取到这种服务。

这样，我们就需要在游戏服务端中提供服务的概念。场景同步服务是一种服务，聊天服务是另一种服务。基于这个思路，我们继续探讨服务应该如何定义，服务有哪些类型，不同类型的服务的消息流应该是怎样的。

3.2 Service-Oriented游戏服务端

3.2.1 游戏服务端中的服务

定义问题

之前提到，传统MMO架构随发展逐渐出现了分拆的需求，最常见的是把聊天逻辑从全局进程中拆出来。

这种拆分的思路是符合service-oriented的发展趋势的，仔细想想的话，其实聊天服务本来就应该是具体项目无关的。游戏中可以嵌入公司的公共聊天服务，甚至是第三方提供的聊天服务，比如网易最近开推的云信。http://netease.im/

这样，聊天服务就是独立于游戏业务而持久存在的，我们就从代码复用的层次上升到了服务复用。诚然，公司内不同项目，也可以直接用同一套聊天服务代码库，达到代码级别的复用。但是这样做最后的结果往往就是，每个团队都会从更早的团队拿过来聊天业务代码，然后自己改造改造，成了完全不同的分支，最后连代码复用都做不到了。

从另一个思路来讲，同一款游戏的不同组服务器，其实也只需要同样的一组聊天服务。但是如果按传统的模式，一组服务器只能开零或一个聊天服务器，事实上，有可能某10组服务器用1个聊天服务器就够了，而某1组服务器用1个聊天服务器压力都有些大。

因此，我们可以定义服务的概念。

在明确这个定义之前，你也许注意到了，我在文章的之前部分措词很混乱——一会儿是XX进程，一会儿

是XX服务器，一会儿又是XX服务。现在，我们统称为XX服务（或XXservice）。

比如，场景服务与切场景服务，聊天服务，公会服务等等。

服务是什么？

可以简单理解为一组方法集合。服务是分布式游戏服务端中的最小实体，一个服务提供了一组确定的、可供调用的方法。

skynet中，一个skynet_context唯一对应一个服务，而一个skynet节点对应一组服务；传统MMO中，一个进程对应一组服务，但是很难在其中找到“一个”服务的划分界限。

在确定如何划分服务之前，首先看看服务的类型。

对于游戏服务端的需求来说，服务可以大概分为两类：一类是具有独立命名空间的；一类是在全局命名空间的。

服务的命名空间其实也算是具有游戏开发特色的，虽然不知道最早MMO分服的具体原因是什么，但是就事实而言，ARPG游戏的分服已经成了策划需求。后来又是各种渠道服的需求出现，命名空间更是没办法丢掉。而且还有一点，就是开发阶段本地调试对隔离服务端环境的需求。

举个例子，之前提到的聊天服务就是一种全局命名空间的服务，而对于分服游戏来说，场景服务就是具有独立命名空间的服务。而对于手游来说，可以划分出的服务就更多了。

服务划分解决了什么问题？

以进程为单位开发和服务为单位开发是两种不同的思路。人是有惰性的，如果不是特别必要，上面也没人强推，那我想大部分程序员还是会把聊天服务实现在全局协调进程里。

服务的概念就是为了提出一种与物理容器无关的抽象。服务可以以某个进程为容器，也可以以某个线程为容器。可以像skynet一样以一个luaState为容器，也可以像Erlang游戏服务端那样以一个actor为容器。而一个容器也可以提供多种服务。

注意，这里提出的服务这种抽象与之前所说的Gate这种基础设施抽象是不同的。如果将游戏服务端看做一个整体，那么Gate就是其中的static parts，服务就是其中的dynamic parts。两者解决的是不同层面的问题。

服务划分的核心原则是将两组耦合性较低的逻辑划分为不同的服务。具体实现中肯定不存在完美的划分方案，因此作为让步，只要是互交互不多的逻辑都可以划分为不同的服务。举一个简单的例子就是公会服务v.s.场景服务，两者的关系并不是特别密切。

引入新的问题

服务划分的极端是每一个协议包都对应一种服务。事实上，服务的定义本来就是基本隔绝的逻辑集合。如果服务定义得太多，服务间数据交互就会复杂到程序员无法维护的程度。

复杂数据交互的另一方面，是复杂的网络拓扑。基于我们之前的架构，client与服务的通信可以借助Gate简化模型，但是服务之间的通信却需要 O(n^2)的连接数。服务都是dynamic parts，却对其他服务

的有无产生了依赖，而且大部分情况下这种依赖都是双向的。整个服务端的网络会错综复杂。

3.2.2 游戏服务端中的Message Queue

定义问题

我们要解决的最关键的问题是：如果服务之间很容易就产生相互依赖，应该如何化简复杂的网络拓扑。如果说得实际一点，那就是让服务器组的启动流程与关闭流程更加优雅，同时保证正确性。

skynet给我带来的思路是，服务与服务之间无需保持物理连接，而只需要借助自己寄宿的skynet与其他服务通信，相当于所有服务间的连接都是抽象的、虚拟的。skynet是整个集群中的static parts，服务作为dynamic parts启动顺序肯定在skynet之后。

skynet可以大大简化服务间拓扑关系，但是其定位毕竟不在于此，比如，skynet并不做消息的qos保证，skynet也没有提供各种方便的外围设施。我们还需要提供更强大语义的基础设施抽象。

面对这种需求，我们需要一种消息队列中间件。

生产者消费者一直都是一种比较经典的解耦模型，而消息队列就是基于这种模型构建的。每个skynet节点本质上就是一个高度精简的消息队列，为寄宿的每个服务维护一个私有队列，对全局队列中的消息dispatch，驱动寄宿服务。

而我希望的是更纯粹的消息队列中间件，选择有很多，下面以RabbitMQ为例简单介绍。

RabbitMQ提供了消息队列中间件能提供的所有基本语义，比如消息的ack机制和confirm机制、qos保证、各种pattern的支持、权限控制、集群、高可用、甚至是现成的图形化监控等等。接下来的讨论会尽量不涉及RabbitMQ具体细节，把它当做一个通常的消息队列中间件来集成到我们目前为止形成的游戏服务端之中。当然，Gate经过扩展之后也能替代MQ，但是这样就失去了其作为Gate的意义——Gate更多的是用在性能敏感的场合，比如移动同步，协议太重是没有必要的。而且，重新造个MQ的轮子，说实话意义真的不大。

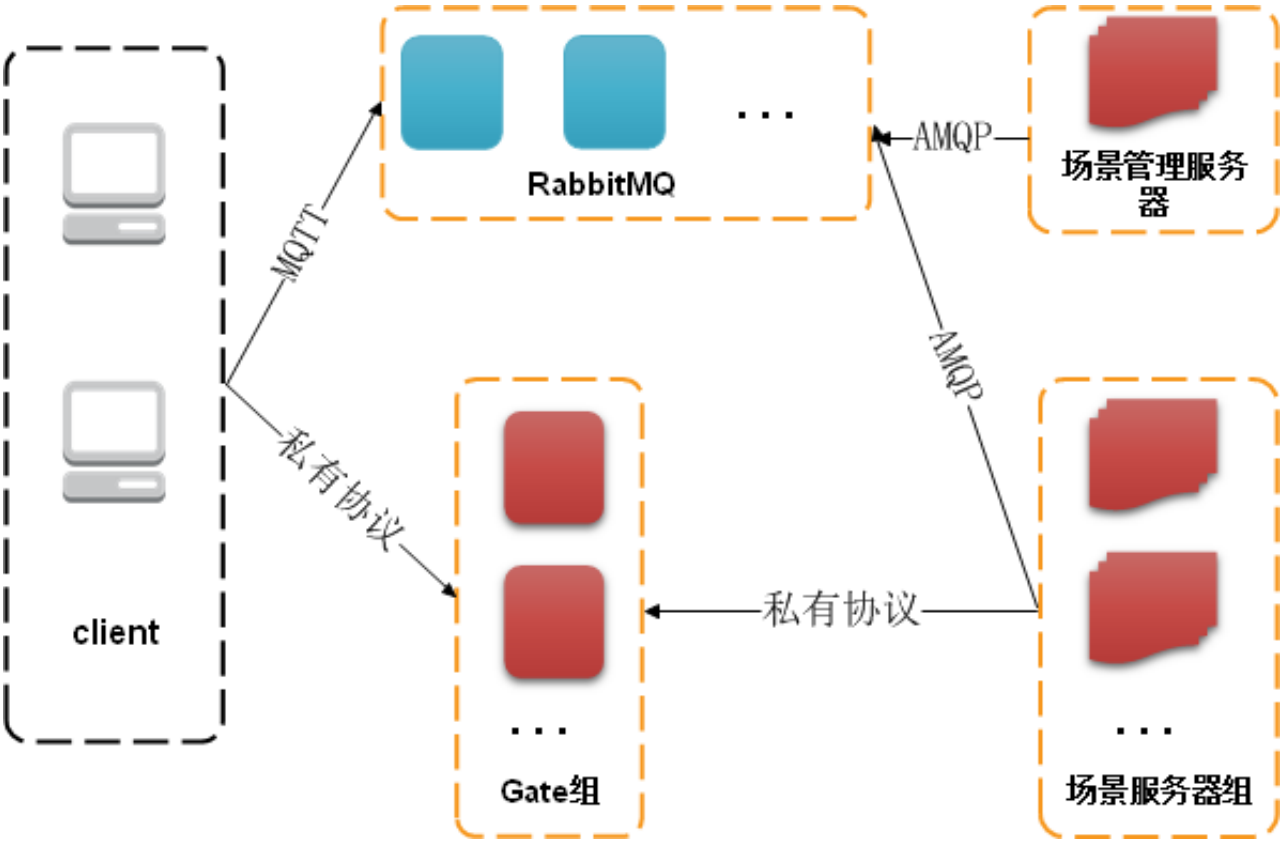
MQ解决了什么问题？

MQ与Gate的定位类似，都是整个生态中的static parts。但是MQ与Gate是两种不同的基础设施抽象，提供的语义也不尽相同。

- Gate要解决的问题是以最低的成本构建消息流模型，仅提供传输层所能提供的消息送到质量保证（TCP撕UDP暂时领先）。client与Gate的连接断开，Gate没有义务再保留连接上下文。Gate其实是在游戏场景同步需求情景下诞生的特殊的MQ，具有一部分MQ的职责，比如发布订阅（客户端发布，服务端订阅，带组播的话还支持message dup），协议高度简化（对比下AMQP协议的复杂度。。），没有一些MQ专有的capability（qos保证，消息持久化等）。
- MQ要解决的问题是提供普适的消息队列抽象。性能不敏感的服务可以依赖MQ构建，将自己的连接维护与会话保持两块状态寄存在MQ上。

这样，我们的服务端中就出现了两个static parts——一个是Gate，一个是MQ。Gate与MQ是两个完全无关的基础设施，这两部分先于其他所有dynamic parts启动、构建。场景服务连接Gate与MQ（前提是确实有其他服务会与场景服务进行通信），聊天服务连接MQ，client连接Gate与MQ。

引入MQ之后的拓扑：



再脑补一下，增加N个节点，就形成了Gate和MQ的双中心，网络拓扑优雅了很多。

就具体实现来说，之所以选择RabbitMQ，还因为其对mqtt协议支持的比较好，官网上就有插件下载。mqtt协议可以参考这里。client走mqtt协议跟MQ通信还是比较轻量级的。

引入新的问题

现在，client或者服务都需要通过不同的协议（Gate、MQ）与其他部分通信。

这样对应用层开发者来说就是一个负担。

- 服务端和客户端，走Gate的流程都是基于私有协议与自有的API。
- 同时，在客户端，走MQ的流程基于mqtt协议与mqttLib的API；在服务端，走MQ的流程基于amqp协议与rabbitMQ的API。

ps. Gate的私有协议和mqtt、amqp协议下面会统称为消息路由协议。

而且不同的API的调用模型都不一样，因此我们需要一种应用层统一的调用规范。

3.3 游戏服务端中的RPC与Pattern

3.3.1 RPC

定义问题

整理一下现状。

目前，client可以发送两类消息：一类交由Gate路由，一类交由MQ路由。service也可以接收两类消息：一类由Gate路由过来，一类由MQ路由过来。

我们希望的是，应用层只需要关心服务，也就是说发送的消息是希望转到哪个服务上，以及接收的消息是请求自己提供的哪个服务。
这样对于应用层来说，其看到的协议应该是统一的，而至于应用层协议的底层协议是Gate的协议还是MQ的协议，由具体的适配器（Adaptor）适配。

这个应用层的协议就是RPC的一部分。
RPC一直都是很有争议的。一方面，它能让代码看起来更优雅，省了不少打解包的重复代码；另一方面，程序员能调RPC了，系统就变得很不可控，特别是像某些架构下面RPC底层会绕很多，最后用的时候完全违背设计本意。

但是总的来说，RPC的优势还是比较明显的，毕竟游戏服务端的整体服务定义都是同一个项目组内做的，副作用严格可控，很少会出现调用一条RPC要绕很多个节点的情况。

RPC解决什么样的问题？

RPC的定位是具体消息路由协议与应用层函数调用的中间层。一个标准的RPC框架要解决两个问题：

- 第一是协议定义。
- 第二是调用规范的确立。

RPC的协议定义也可以做个划分：

- 协议中的一部分用来标识一次调用session，可以用来实现RPC的回调，可以用来实现RPC的超时管理等等。
- 另一部分用来标识调用的具体方法。这部分其实跟用不用RPC没太大关系，因为不用RPC只是打解包的话也是会用一些协议序列化、反序列化协议包的。采用RPC框架的话，就是希望这部分工作尽可能多的自动化。

RPC调用规范的核心设计意图就是让应用层程序员调用起来非常自然、不需要有太多包袱（类bigworld架构的rpc设计通常也是这个原则，尽量让应用层不关注切进程的细节）。调用规范的具体细节就跟语言和平台相关了。在支持异步语法的语言/平台，可以原生集成异步等待、执行完恢复上下文继续执行的语义。在不支持异步语法的语言/平台，那就只能callback。如果是不支持将函数作为参数传递的语言/平台，我想你应该已经离现代游戏开发太远了。

通用的部分确定之后，还得解决特定于具体路由方式的、需要适配的部分。

我将这部分逻辑称为Adaptor，很好理解，就是RPC到具体消息路由协议、具体消息路由协议到RPC的适配器。

下面，结合一种具体的RPC实现方式（下文称为Phial规范），来探讨下如何将上面提出的这几个概念串起来。

先通过一个大概的流程来厘清一次RPC流程中涉及的所有角色。

RPC既然作为一次远程过程调用，那么，对于调用方来说，其调用的是一个跟普通函数很像的函数（有可能表现为一个异步函数，也有可能表现为一个同步函数）；对于被调用方来说，其被调用的就真的是自己的一个函数了。

整个的pipeline也很清晰：

- 调用方调用某个服务的某个函数，RPC层会根据之前说的RPC层协议将调用信息（invokeId、方法id、参数等）打包，并将打包的消息和函数对应服务的路由规则告诉路由适配层，路由适配层根据路由规则给打包消息加个消息头，然后传给路由层（具体的Gate路由或MQ路由）。
- 路由层将消息路由到对应节点，该节点上的路由适配层解出消息头和打包消息，根据消息头确定被请求服务，并这些信息传给RPC层，RPC层解打包消息得到调用信息，然后做一次dispatch，被调用方的一开始注册进来的对应函数就会被回调到了。

在这个过程中，我们称调用方可以调用的是服务的delegate（可以类比为Stub），被调用方注册进来的是服务的implement（可以类比为Skeleton）。路由适配层就是Adaptor。可以基于不同类型的Adaptor构造服务的delegate。服务的implement也可以注册在不同的Adaptor上。不同的Adaptor只需要针对RPC层提供同样的接口，让RPC层可以发送打包消息和服务特定的路由规则，可以注册implement即可保证RPC层与Adaptor层是完全无关的。

我们在示例中实现了多种Adaptor，目前为止涉及到的有MqttAdaptor、GateAdaptor、AmqpAdaptor。

除了这整个的数据流之外，示例中还包装了两种异步调用与回调形式。

- 一种是针对.Net 2.0的callback模式；
- 一种是针对.Net 4.5的Task await/async模式。

第一种专门针对不支持.Net 4.5的平台，比如Unity。但是只要针对这种形式稍加扩展，也能支持.Net 2.0的yield语义，实现简单协程。关于.Net 2.0中的协程实现，可以参考这里。

两种异步方式实现回调的原理是相同的，都是本地hold住调用上下文，等回包的时候检查即可。

这样，在支持.Net 4.5的平台，一个穿插了RPC调用的函数就可以写成这个样子：

```
C#
service.AskXXX(x, y).Callback =
operation =>
{
        if (operation.IsComplete)
        {
                var ret = operation.Result;
        }
};
```

关于一些细节的说明

- 我在之前的流程里面特意没有说明打包协议，打包协议选择很多，比如msgpack、bson、pb、pbc等等。示例实现中采用的是一种顺序的二进制打解包机制，缺点就是没办法做版本兼容，优点就是实现

起来简单速度快。当然换打包协议也是很容易的。示例项目后续会增加对多种打包协议的支持。

- 由于这个RPC框架主要是针对unity游戏，客户端部分与服务端部分的平台本质是不同的，客户端以.Net 2.0为基础，服务端以.Net 4.5为基础。相关的服务定义文件也都隔离成了两个库，既减少了对客户端的协议暴露，又可以保证客户端依赖库的体积最小。
- Adaptor的接口设计。Adaptor是为RPC层服务的，因此不同的Adaptor所需要实现的接口只需要面向RPC层保持一致。Adaptor需要针对delegate与implement提供不同的抽象意义。对于implement来说，Adaptor是一个持续产出消息的流；对于delegate来说，Adaptor是一个可以接受消息的传输器。应用层对自己掌握的Adaptor是知情的，因此Adaptor可以提供特化的接口，比如client需要的所有Adaptor都需要额外提供Poll接口，场景服务需要的GateAdaptor也需要Poll接口。

引入新的问题

有了RPC之后，我们可以在应用层以统一的形式进行服务请求。

但是这样还不够——我们目前所提的RPC就是普通的方法调用，虽然对应用层完全隐藏了协议或者其他中间件的细节，但是这样一来这些中间件的强大特性我们也就无法利用了。

还应该有另一种与RPC平行的抽象来特化RPC的形式，这种抽象与RPC共同组成了一种游戏开发规范。

3.3.2 RPC、Pattern与规范

定义问题

由于不同的中间件解决问题的方式不同，因此我们没办法在应用层用统一的形式引用不同的中间件。因此，我们可以针对游戏开发中的一些比较经典的消息pipeline，定义pattern。然后，用pattern与RPC共同描述服务应该如何声明，如何被调用。

Pattern解决了什么问题

- pattern规定了客户端与服务、服务与服务的有限种交互形式。
- pattern解决了之前我们只能靠感觉确定服务应该走哪种基础设施抽象的问题。

不同的基础设施抽象可以实现不同的pattern子集，如果需要新增加一类基础设施，我们可以看它的功能分别可以映射到哪几种pattern上，这样就能直接集成到Phial规范中。

下面，就针对游戏服务端的常见需求，定义几种pattern。

三种通信情景：

- client -> server

最简单的pattern是ask，也就是向服务发起一次异步调用，然后client不关注服务的处理结果就直接进行后续的逻辑。最常见的就是移动请求。
还有一种是传统MMO中不太重视，而异步交互手游反倒从web引入的request。client向服务发起一次异步调用，但是会等到服务处理结果返回（或超时）才进行后续的逻辑。例子比较多，比如一次抽卡或者一次异步PVP。

- server -> client

与ask对应的是sync，是服务进行一次无源的对client的impl调用，client无条件执行impl逻辑。sync需要指明被调用方。这种最常见的是移动同步。有一点需要注意，示例中实现了一种形式比较丑陋的组播sync，依赖了Gate的私有协议，也就是forward指定一个int值。这个之后会做调整。

与request对应的是reply。这个相当于是处理一次request然后直接返回一个值，没什么特别之处。

- server -> server

最常见的就是invoke，相当于一次希望返回值的远程调用。例子有很多，适用于任意两个服务间的通信需求。

还有一种解耦利器我称之为notify。当然本质上其实就是pub-sub，消息会在中间件上dup。应用情景是消息提供者/事件源只管raise event，而不关注event是否被处理、event后续会被路由到哪里。在中间件上，只要有服务实现了该notify service的impl，就能得到通知；如果没有任何节点提供对该服务的impl，就相当于消息被推到了sink。应用情景是可以将玩家行为log以及各种监控、统计系统逻辑从业务代码中剥离出来，事件源触发的逻辑只有一处，而处理的逻辑可以分散在其他监控进程中，不需要增加一种监控就得在每个事件源都对应插一行代码。

Gate和MQ实现了不同的pattern集合。当然，正如之前所说，Gate本质上也是一种MQ，但是由于我们对这两种基础设施抽象的定位不同，所以在实现各自的Adaptor的时候也限定了各自支持的pattern。比如，Gate不能支持notify，MQ不能支持ask-sync。

我在示例实现中没有加入MQ对客户端组播的支持。主要原因是考虑到client是通过MQTT协议跟MQ通信，相当于组维护是client发起的。对于聊天这种的可能还好，对于其他的可能会有隐患。

引入新的问题

到目前为止，我们总结出了如下几种与游戏服务端有关的消息pipeline：

- client -> Gate -> service
- service -> Gate -> client
- service -> Gate -> client*

- service -> MQ -> service
- service -> MQ -> service*

这基本上已经能涵盖游戏中的大部分需求了，因此我们对消息流的讨论就到此为止。

接下来讨论游戏世界的状态维护。

维护游戏世界状态的职责同样由一种服务负责，这种服务下面称为数据服务。

但是数据服务所说的服务与之前所提的作为dynamic parts的Phial服务不太相同，实际上是一些基础设施抽象和Phial服务的组合。

有了数据服务，我们还可以更加明确client与service走Gate与MQ究竟有什么本质区别。

4 游戏世界状态的维护方式

4.1数据服务的定位

游戏世界的状态可以简单分为两个部分，一部分是需要存档的，比如玩家数据；一部分是不需要存档的，比如场景状态。

对于访问较频繁的部分，比如场景状态，会维护成纯内存数据；对于访问较不频繁的部分，比如玩家存档，就可以考虑维护在第三方。这个第三方，就是数据服务。

数据服务与之前所提到的场景服务、IM服务等都属于应用层的概念。数据服务通常也会依赖于一种基础设施抽象，那就是缓存。

4.1.1 传统架构中的数据服务

传统MMO架构中，数据服务的概念非常模糊。

我们还是先通过回顾发展历史的形式来厘清数据服务的定义。回到场景进程的发展阶段，玩家状态是内存中的数据，但是服务器不会一直开着，因此就有了存盘（文件或db）需求。但是随着业务变复杂，存盘逻辑需要数据层暴露越来越多的存储API细节，非常难扩展。因此发展出了Db代理进程，场景进程直接将存档推给Db代理进程，由Db代理进程定期存盘。
这样，存储API的细节在Db代理进程内部闭合，游戏逻辑无须再关注。场景进程只需要通过协议封包或者RPC的形式与Db代理进程交互，其他的就不用管了。
Db代理进程由于是定期存盘，因此它相当于维护了玩家存档的缓存。这个时候，Db代理进程就具有了数据服务的雏形。

跟之前的讨论一样，我在这里又要开始批判一番了。

很多团队至今，新立项的项目都仍然采用这种Db代理进程。虽然确实可以用来满足一定程度的需求，但是，存在几个致命问题。

- 第一，Db代理进程让整个团队的代码复用级别保持在copy-paste层面。玩家存档一定是项目特定的，而采用Db代理进程的团队，通常并不会将Db代理进程设计成普适、通用的，毕竟对于他们来说，Db代理进程是场景进程和存盘之间的唯一中间层。举个例子，Db代理进程提供一个LoadPlayer的RPC接口，那么，接口实现就一定是具体游戏相关的。
- 第二，Db代理进程严重耦合了两个概念：一个是面向游戏逻辑的存储API；一个是数据缓存。数据缓存本质上是一种新的基础设施抽象，kv发展了这么多年，已经涌现出无数高度成熟的工业级缓存基础设施，居然还有新立项游戏对此后知后觉。殊不知，自己对Db代理进程再怎么做扩展，也不过是在feature set上逐渐接近成熟的KV，但是在可用性上就是玩具和工业级生产资料的差距。举个最简单的例子，有多少团队的Db代理进程能提供一个规范化的容忍多少秒掉线的保证？
- 第三，Db代理进程在分区分服架构下通常是一区一个的，一个很重要的原因就是Db代理进程通常是自己YY写出来的，很少能够解决扩容问题。如果多服共用一个Db代理进程，全局单点给系统增加不稳定

性的问题暂且按下不表，负载早就撑爆了。但是只是负责缓存玩家存档以及将存档存盘，这跟之前讨论过的全局IM服务定位非常类似，又有什么必要分区分服？

我们可以构建一个数据服务解决这些问题。至于依赖的具体缓存基础设施，我之后会以redis为例。

redis相比于传统的KV比如memcache、tc，具有不同的设计理念，redis的定位是一种数据结构服务器。游戏服务端开发可以拿redis当缓存用，也可以直接当一个数据库用。

数据服务解决了什么问题

数据服务首先要解决的就是玩家存档问题。redis作为一个高性能缓存基础设施，可以满足逻辑层的存档需求。同时还可以实现额外的落地服务，比如将redis中的数据定期存回mysql。之所以这样做，一方面是因为redis的定位是高性能缓存设施，那就不希望它被rdb、aofrewrite机制拖慢表现，或者卡IO；另一方面是对于一些数据分析系统，用SQL来描述数据查询需求更合适，如果只用redis，还得单独开发查询工具，得不偿失。

数据服务其次要解决的问题是可以做到服务级别的复用。这一点我们可以借助企业应用开发中的ORM来设计一套对象-kv-关系映射。也就是数据服务是统一的，而不同的业务可以用不同的数据结构描述自己的领域模型，然后数据服务的配套工具会自动生成数据访问层API、redis中cache关系以及mysql中的table schema。也就是说，同样的数据服务，我在项目A中引用并定义了Player结构，就会自动生成LoadPlayer的API；在项目B中定义User同理生成LoadUser的API。

这两个问题是比较容易解决的，最关键的还是一个思路的转换。

实际上，数据服务除去缓存基础设施的部分，都属于外围机制。在有些设计中，我们可以看到还是存在缓存服务与逻辑服务的中间层。这种中间层的单点问题很容易解决——只要不同的逻辑服务访问不同的中间层节点即可。中间层的意义通常是进行RPC到具体缓存协议API的转换，在我的实现中，由于已经有了数据访问API的自动生成，因此没有这种中间层存在的必要。所有需要访问数据服务的逻辑服务都可以直接通过数据访问API访问。

其中还有几点细节：

- 数据访问层API的调用规范与RPC的调用规范保持了统一，都是基于async/await模式。
- 通过数据服务对任意存档进行增加或修改都会记录一个job，由落地服务定期检查job进行落地。

引入新的问题

目前仍然遗留了几个问题：

- redis单实例的性能确实很强悍，但是如果全区全服只开一个redis实例确实是存在问题的，这个问题需要解决。
- 数据服务对于传统MMO架构来说可以无缝替换掉丑陋的Db代理进程，但是，既然数据服务已经能提供抽象程度如此高的存储接口，那是否还可以应用在其他地方？

4.1.2 无状态服务中数据服务的定位

定义问题

之前提到过，游戏世界的状态除了需要存档的玩家数据，还有一部分是不需要存档的逻辑服务的状态。

数据服务如果只是用来替代MMO中的Db代理进程的，那么它的全部职责就仅仅是为需要存档的数据提供服务。从更高的抽象层次来看的话，数据服务相当于是维护了client在服务端的状态。

但是，数据服务提供了更强大的抽象能力。现在数据服务的API结构是任意定制的、code first，而且数据服务依赖的基础设施——redis又被证明非常强大，不仅仅是性能极佳，而且提供了多种数据结构抽象。那么，数据服务是否可以维护其他服务的状态？

在web开发中，用缓存维护服务状态是一种很常规的开发思路。而在游戏服务端开发中，由于场景服务的存在，这种思路通常并不靠谱。

为什么要用缓存维护服务状态？

考虑这样一个问题：如果服务的状态维护在服务进程中，那么服务进程挂掉，状态就不存在了。而对于我们来说，服务的状态是比服务进程本身更加重要的——因为进程挂了可以赶紧重启，哪怕耽误个1、2s，但是状态没了却意味着这个服务在整个分布式服务端中所处的全局一致性已经不正确了，即使瞬间就重启好了也没用。
那么为了让服务进程挂掉时不会导致服务状态丢掉，只要分离服务进程的生命周期和服务状态的生命周期就可以了。

将进程和状态的生命周期分离带来的另一个好处就是让这类服务的横向扩展成本降到最低。

比较简单的分离方法是将服务状态维护在共享内存里——事实上很多项目也确实是这样做的。但是这种做法扩展性不强，比如很难跨物理机，而且共享内存就这样一个文件安全性很难保障。

我们可以将服务状态存放在外部设施中，比如数据服务。

这种可以将状态存放在外部设施的服务就是无状态服务（stateless service）。而与之对应的，场景服务这种状态需要在进程内维护的就是有状态服务（stateful service）。

有时候跟只接触过游戏服务端开发的业务狗谈起无状态服务，对方竟然会产生 一种“无状态服务是为了解决游戏断线重连的吧”这种论点，真的很哭笑不得。断线重连在游戏开发中固然是大坑之一，但是解决方案从来都跟有无状态毫无关系， 无状态服务毕竟是服务而不是客户端。如果真的能实现一个无状态游戏客户端，那真的是能直接解决坑人无数的断线重连问题。

无状态游戏客户端意味着网络通信的成本跟内存数据访问的成本一样低——这当然是不可能实现的。

无状态服务就是为了scalability而出现的，无状态服务横向扩展的能力相比于有状态服务大大增强，同时实现负载均衡的成本又远低于有状态服务。

分布式系统中有一个基本的CAP原理，也就是一致性C、响应性能A、分区容错P，无法三者兼顾。无状态服务更倾向于CP，有状态服务更倾向于AP。但是要补充一点，有状态服务的P与无状态服务的P所能达到的程度是不一样的，后者是真的容错，前者只能做到不把鸡蛋放在一个篮子里。

两种服务的设计意图不同。无状态服务的所有状态访问与修改都增加了内网时延，这对于场景服务这种

性能优先的服务是不可忍受的。而有状态服务非常适合场景同步与交互这种数据密集的情景，一方面是数据交互的延迟仅仅是进程内方法调用的开销，另一方面由于数据局部性原理，对同样数据的访问非常快。

既然设计意图本来就是不同的，我们这一节就只讨论数据服务与无状态服务的关系。

游戏中可以拆分为无状态服务的业务需求其实有很多，基本上所有服务间交互需求都可以实现为无状态服务。比如切场景服务，因为切场景的请求是有限的，对时延的要求也不会特别高，同理的还有分配房间服务；或者是面向客户端的IM服务、拍卖行服务等等。

数据服务对于无状态服务来说，解决了什么问题？

简单来说，就是转移了无状态服务的状态维护成本，同时让无状态服务具有了横向扩展的能力。因为状态维护在数据服务中，所以无状态服务开多少个都无所谓。因此无状态服务非常适合计算密集的业务需求。

你可能觉得我之前在服务划分一节之后直接提出要引入MQ有些突兀，实际上，服务划分要解决的根本问题就是让程序员能清楚自己定义每种服务的意图是什么，哪一种服务更适合Request-Reply，哪一种服务更适合Ask-Sync。

假设策划对游戏没有分服的需求，理论上讲，有节操的程序是不应该以“其他游戏就这样做的”或“做不到”之类的借口搪塞。每一种服务都由分布式的多个节点共同提供服务，如果服务的消息流更适合Request-Reply pattern，那么实现为无状态服务就更合适，原因有二：

- 一个Request上来，取相关数据，处理，直接返回。整个状态的生命周期保持在一次RPC调用过程中，这描述的就是Request-Reply的工作方式。
- 目前只有走MQ的消息pipeline支持Request-Reply pattern，而MQ通常都能很好地支持无状态服务的round-robin work distribution。

针对第二点，可能需要稍微介绍下rabbitMQ。rabbitMQ中有exchange（交换机）、queue、binding（绑定规则）三个主要概念。其中，exchange是对应生产者的，queue是对应消费者的，binding则是描述消息从exchange到queue的路由关系的。exchange有两种常用类型direct、topic。其中direct exchange接收到的消息是不会dup的，而topic exchange则会将接收到的消息根据匹配的binding确定要dup到哪个target queue上。

这样，对于无状态服务，比如同一命名空间下的切场景服务，可以共用同一个queue，然后client发来的消息走direct exchange，就可以在MQ层面做到round-robin，将消息轮流分配到不同的切场景服务上。而且无状态服务本质上是没有扩容成本的，波峰就多开，波谷就少开。

程序员负责为不同服务规划不同的横向扩展方式。比如类似公会服务这种走MQ的，横向扩展的触发条件就是现在请求数量级或者是节点压力。比如场景服务这种Ask-Sync的，横向扩展就需要借助第三方的服务作为仲裁者，而这个仲裁者可以实现为基于MQ的服务。

这里有个问题需要注意一下。

由于现在同一个client上来的request消息可能由无状态服务的不同节点处理，那么就会出现这样的情

况：

1. 某个client由于一些原因，快速发了两个message1、message2。
2. message1先到了服务A，服务A去数据服务拉相关数据集合Sa，并进行后续处理。
3. 此时message2到了服务B，服务B去数据服务拉相关数据集合Sb，进行后续处理，处理完毕，将结果存回数据服务。
4. 然后服务A才处理完，并尝试将处理结果存回数据服务。

假如Sa与Sb有交集，那就会出现竞态条件，如果这时允许服务A存回结果，那数据就有可能存在不一致。

类似的情况还会出现在像率土之滨或者cok这种策略游戏的大世界刷怪需求中。当然前提是玩家与大地图上的元素交互和后台刷怪逻辑都是基于无状态服务做的。

这其实是一个跨进程共享状态问题，而且是一个高度简化的版本——因为这个共享状态只在一个实例上维护。可以引入锁来解决问题，思路通常有两个：

最直观的一种方案是悲观锁。也就是如果要进行修改操作，就需要在读相关数据的时候就都加上锁，最后写成功的时候释放锁。获得锁所有权期间其他impure服务任意读写请求都是非法的。

但是，这毕竟不是多线程执行环境，没有语言或平台帮你做自动锁释放的保证。获取悲观锁的服务节点不能保证一定会将锁释放掉，拿到锁之后节点挂掉的可能性非常大。这样，就需要给悲观锁增加超时机制。

第二种方案是乐观锁。也就是impure服务可以随意进行读请求，读到的数据会额外带个版本号，等写的时候对比版本号，如果一致就可以成功写回，否则就通知到应用层失败，由应用层决定后续操作。

带过期机制的悲观锁和乐观锁本质上都属于可抢占的分布式锁，相当于是将paxos要解决的问题退化为单Acceptor，因此实现起来非常简单。可过期的悲观锁和乐观锁唯一的区别就是前者在申请锁的时候有可能申请失败，而后者申请锁时永远不会失败。两种方案具体的表现优劣跟业务需求有关，不论一开始选择的是哪一种，都非常容易切换到另一种。

我在示例中实现了一个简单的乐观锁，在提交修改的时候用一个lua脚本做原子检查就能简单实现。如果要实现带过期机制的悲观锁，需要保证应用层有简单的时钟同步机制，而且在申请锁的时候也要写一个lua脚本。

在应用层也做了对应修改，调用数据访问层API可以按如下这种方式调用。之所以用了RTTI，是考虑到有可能会改成悲观锁实现，在Dispose的时候会自动release lock。现在pure服务与impure服务对数据服务调用的接口是不一样的，我们甚至还可以基于这一点在底层做一些扩展，最典型的比如读写分离。当然，这些都是引入主从之后要考虑的问题了。

C#

```
using (var structFstAccesser = await GetStructFstAccesser())
{
        using (var structSndAccesser = await GetStructSndAccesser())
        {
```

```
                var fieldFst = await structFstAccesser.LoadFieldFstAsync();
                var fieldSnd = await structSndAccesser.LoadFieldSndAsync();
                // logic here...
                structFstAccesser.UpdateFieldFst(fieldFst);
                structSndAccesser.UpdateFieldSnd(fieldSnd);
                await SubmitChanges(structFstAccesser, structSndAccesser);
                // result handle here
                return true;
        }
}
```

有了这样一个简易的锁机制，我们可以保证单redis实例内的一致性。

引入新的问题

有了无状态服务的概念，我们的架构中就可以逐步干掉类似切场景管理这种单点进程。无状态服务是高可用的，也就是说，任意挂掉一个，仍然能持续提供服务。

整个游戏服务端理论上应该具有整体持续提供服务的能力。也就是说，随便挂掉一个节点，不需要停服。场景服务挂掉一个节点，不会影响其他任何服务，只是玩家短期内无法进行场景相关操作了而已。

而我们见过的大多数架构，处处皆单点，这完全不能叫可用的架构。有的时候一个服务端跑的好好的，有人硬是要额外加一个全局单点，而且理由是更容易管理，让人哭笑不得。分布式系统中动不动就想加单点，这是病，得治。判断一整个游戏服务端是否具有可用性很简单，随便kill掉一个节点，如果服务端仍然能持续提供服务，即使是部分client受到了影响，也能称为是可用的。

但是，现在逻辑服务具有可用性了，可是数据服务还没有具有可用性，数据服务依赖于一个redis实例，这个redis实例反而成为了整个服务端中的单点。

幸好，redis像其他大多数工业级缓存基础设施一样，已经提供了足够用的可用性机制。但是，在讨论redis的可用性机制之前，我们先解决一下数据服务的一个遗留问题，那就是如何构建一个可以扩展的全局数据服务。

4.2 数据服务的扩展

redis是一种stateful service，继续应用之前的CAP原则，redis是倾向于AP的。之后我们可以看到，redis的各种扩展，实际上都是基于这个原则来做的。

4.2.1 分片方案

定义问题

我们遇到的问题是，如果将数据服务定位为全局服务，那仅用单实例的redis就难以应对多变的负载情况。毕竟redis是单线程的。

从mysql一路用过来的同学这时都会习惯性地水平拆分，redis中也是类似的原理，将整体的数据进行切分，每一部分是一个分片shard，不同的shard维护的key集合是不同的。

那么，问题的实质就是如何基于多个redis实例设计全局统一的数据服务。同时，有一个约束条件，那就是我们为了性能需要牺牲全局一致性。也就是说，数据服务进行分片扩展的前提是，不提供跨分片事务的保障。redis cluster也没有提供类似支持，因为分布式事务本来就跟redis的定位是有冲突的。

因此，我们之后的讨论会有一个预设前提：不同shard中的数据一定是严格隔离的，比如是不同组服的数据，或者是完全不相干的数据。要想实现跨shard的数据交互，必须依赖更上层的协调机制保证，底层不做任何承诺。

这样，我们的分片数据服务就能通过之前提到的简易锁机制提供单片内的一致性保证，而不再提供全局的一致性保证。

基于同样的原因，我们的分片方案也不会在分片间做类似分布式存储系统的数据冗余机制。

分片方案解决了什么问题

分片需要解决两个问题：

- 第一个问题，分片方案需要描述shard与shard之间的联系，也就是cluster membership。
- 第二个问题，分片方案需要描述dbClient的一个请求应该交给哪个shard，也就是work distribution。

针对第一个问题，解决方案通常有三：

- presharding，也就是sharding静态配置。
- gossip protocol，其实就是redis cluster采用的方案。简单地说就是集群中每个节点会由于网络分化、节点抖动等原因而具有不同的集群全局视图。节点之间通过gossip protocol进行节点信息共享。这种方案更强调CAP中的A原则，因为不需要有仲裁者。
- consensus system，这种方案跟上一种正相反，更强调CAP中的C原则，就是借助分布式系统中的仲裁者来决定集群中各节点的身份。

需求决定解决方案，对于游戏服务端来说，后两者的成本太高，而且增加了很多不确定的复杂性，因此现阶段这两种方案并不是合适的选择。比如gossip protocol，redis cluster现在都不算是release，确实不太适合游戏服务端。而且，游戏服务端毕竟不是web服务，通常是可以在设计阶段确定每个分片的容量上限的，也不需要太复杂的机制支持。

但是第一种方案的缺点也很明显，做不到动态增容减容，而且无法高可用。但是如果稍加改造，就足以满足需求了。

在谈具体的改造措施之前，先看之前提出的第二个问题。

第二个问题实际上是从另一种维度看分片，解决方案很多，但是如果从对架构的影响上来看，大概分为两种：

- 一种是proxy-based，基于额外的转发代理。例子有twemproxy/Codis。
- 一种是client sharding，也就是dbClient（每个对数据服务有需求的服务）维护sharding规则，自助式选择要去哪个redis实例。redis cluster本质上就属于这种，client侧缓存了部分sharding信

息。

第一种方案的缺点显而易见，在整个架构中增加了额外的间接层，pipeline中增加了一趟round-trip。如果是像twemproxy或者Codis这种支持高可用的还好，但是github上随便一翻还能找到特别多的没法做到高可用的proxy-based方案，无缘无故多个单点，这样就完全搞不明白sharding的意义何在了。

第二种方案的缺点就是集群状态发生变化的时候没法即时通知到dbClient。

第一种方案，我们其实可以直接pass掉了。因为这种方案本质上还是更适合web开发的。web开发部门众多，开发数据服务的部门有可能和业务部门相去甚远，因此需要统一的转发代理服务。但是游戏开发不一样，数据服务逻辑服务都是一帮人开发的，没什么增加额外中间层的必要。

那么，看起来只能选择第二种方案了。

将presharding与client sharding结合起来后，现在我们的改造成果是：数据服务是全局的，redis可以开多个实例，不相干的数据需要到不同的shard上存取，dbClient掌握这个映射关系。

引入新的问题

目前的方案只能满足游戏对数据服务的基本需求。

大部分采用redis的游戏团队，一般最终会选定这个方案作为自己的数据服务。后续的扩展其实对他们来说不是不可以做，但是可能有维护上的复杂性与不确定性。今天这篇文章，我就继续对数据服务做扩展，后面的内容权当抛砖引玉。

现在的这个方案存在两个问题：

- 首先，虽然我们没有支持在线数据迁移的必要，但是离线数据迁移是必须得有的，毕竟presharding做不到万无一失。而在这个方案中，如果用单纯的哈希算法，增加一个shard会导致原先的key到shard的对应关系变得非常乱，抬高数据迁移成本。
- 其次，分片方案固然可以将整个数据服务的崩溃风险分散在不同shard中，比如相比于不分片的数据服务，一台机器挂掉了，只影响到一部分玩家。但是，我们理应可以对数据服务做更深入的扩展，让其可用程度更强。

针对第一个问题，处理方式跟proxy-based采用的处理方式没太大区别，由于目前的数据服务方案比较简单，采用一致性哈希即可。或者采用一种比较简单的两段映射，第一段是静态的固定哈希，第二段是动态的可配置map。前者通过算法，后者通过map配置维护的方式，都能最小化影响到的key集合。

而对于第二个问题，实际上就是上一节末提到的数据服务可用性问题。

4.2.2 可用性方案

定义问题

讨论数据服务的可用性之前，我们首先看redis的可用性。

对于redis来说，可用性的本质是什么？其实就是redis实例挂掉之后可以有后备节点顶上。

redis通过两种机制支持这一点。

- 一种机制是replication。通常的replication方案主要分为两种。一种是active-passive，也就是active节点先修改自身状态，然后写统一持久化log，然后passive节点读log跟进状态。另一种是active-active，写请求统一写到持久化log，然后每个active节点自动同步log进度。

还是由于CAP原则，redis的replication方案采用的是一种一致性较弱的active-passive方案。也就是master自身维护log，将log向其他slave同步，master挂掉有可能导致部分log丢失，client写完master即可收到成功返回，是一种异步replication。
这个机制只能解决节点数据冗余的问题，redis要具有可用性就还得解决redis实例挂掉让备胎自动顶上的问题，毕竟由人肉去监控master状态再人肉切换是不现实的。 因此还需要第二种机制。

- 第二种机制是redis自带的能够自动化fail-over的redis sentinel。reds sentinel实际上是一种特殊的reds实例，其本身就是一种高可用服务，可以多开，可以自动服务发现（基于redis内置的pub-sub支持，sentinel并没有禁用掉pub-sub的command map），可以自主leader election（基于sentinel实现的raft算法），然后在发现master挂掉时由leader发起fail-over，并将掉线后再上线的master降为新master的slave。

redis基于自带的这两种机制，已经能够实现一定程度的可用性。那么接下来，我们来看数据服务如何高可用。

数据服务具有可用性的本质是什么？除了能实现redis可用性的需求——redis实例数据冗余、故障自动切换之外，还需要将切换的消息通知到每个dbClient。

由于是redis sentinel负责主从切换，因此最自然的想法就是问sentinel请求当前节点主从连接信息。但是redis sentinel本身也是redis实例，数量也是动态的，redis sentinel的连接信息不仅在配置上成了一个难题，动态更新时也会有各种问题。而且，redis sentinel本质上是整个服务端的static parts（要像dbClient提供服务），但是却依赖于redis的启动，并不是特别优雅。另一方面，dbClient要想问redis sentinel要到当前连接信息，只能依赖其内置的pub-sub机制。redis的pub-sub只是一个简单的消息分发，没有消息持久化，因此需要轮询式的请求连接信息模型。

上一节末提到过，要想最小化数据迁移成本可以采用两段映射或一致性哈希。这时还有另一种可以扩展的思路，如果采用两段映射，那么我们可以动态下发第二段的配置数据；如果采用一致性哈希，那么我们可以动态下发分片的连接信息。这其中的动态，就可以基于新的符合Phial规范的服务来做。而这个通知机制，就非常适合采用Phial中的Notify pattern实现。而且redis sentinel的实现难度比较低，我们完全可以以较低的成本实现一个扩展性更强，定制性更强，还能额外支持分片服务的部分在线数据迁移机制的服务。
同时，有一部分我在这篇文章里也没提过，那就是落地服务所依赖的mysql的可用性保障机制。相比于再开一个额外的mysql高可用组件，倒不如整合到同样的一个数据服务监控服务中。

这个监控服务就是watcher。由于原理类似，接下来的讨论就不再涉及对mysql的监控部分，只针对redis的。

watcher解决了什么问题？

- 要能够监控redis的生存状态。这一点实现起来很简单，定期的PING redis实例即可。需要的信息以及做出客观下线和主观下线的判断依据都可以直接照搬sentinel实现。
- 要做到自主服务发现，包括其他watcher的发现与所监控的master-slave组中的新节点的发现。前者基于MQ定期Notify通知，后者定期INFO 监控的master实例即可。
- 要在发现master客观下线的时候选出leader进行后续的故障转移流程。这部分实现起来算是最复杂的部分，接下来会集中讨论。
- 选出leader之后将一个最合适的slave提升为master，然后等老的master再上线了就把它降级为新master的slave。

解决这些问题，watcher的职责就已经达成，我们的数据服务也就更加健壮，可用程度更高。

引入新的问题

但是，如果我们引入了新的服务，那就引入了新的不确定性。如果引入这个服务的同时还要保证数据服务具有可用性，那我们就还得保证这个服务本身是可用的。

先简单介绍一下redis sentinel的可用性是如何做到的。同时监控同一组主从的sentinel可以有多个，master挂掉的时候，这些sentinel会根据一种raft算法的工业级实现选举出leader，算法流程也不是特别复杂，至少比paxos简单多了。所有sentinel都是follower，判断出master客观下线的sentinel会升级成candidate同时向其他follower拉票，所有follower同一epoch内只能投给第一个向自己拉票的candidate。在具体表现中，通常一两个epoch就能保证形成多数派，选出leader。有了leader，后面再对redis做SLAVEOF的时候就容易多了。

如果想用watcher取代sentinel，最复杂的实现细节可能就是这部分逻辑了。
这部分逻辑说白了就是要在分布式系统中维护一个一致状态，举个例子，可以将“谁是leader”这个概念当作一个状态量，由分布式系统中的身份相等的几个节点共同维护，既然谁都有可能修改这个变量，那究竟谁的修改才奏效呢？

幸好，针对这种常见的问题情景，我们有现成的基础设施抽象可以解决。

这种基础设施就是分布式系统的协调器组件（coordinator），老牌的有zookeeper（zab），新一点的有etcd（raft）。这种组件通常没有重复开发的必要，像paxos这种算法理解起来都得老半天，实现起来的细节数量级更是难以想象。因此很多现成的开源项目都是依赖这两者实现高可用的，比如codis就是用的zk。

zk解决了什么问题？

就我们的游戏服务端需求来说，zk可以用来选leader，还可以用来维护dbClient的配置数据——dbClient直接去找zk要数据就行了。

zk的具体原理我就不再介绍了，具体的可以参考lamport的paxos paper，没时间没精力的话搜一下看看zk实现原理的博客就行了。

简单介绍下如何基于zk实现leader election。zk提供了一个类似于os文件系统的目录结构，目录结构上的每个节点都有类型的概念同时可以存储一些数据。zk还提供了一次性触发的watch机制。leader election就是基于这几点概念实现的。

假设有某个目录节点/election，watcher1启动的时候在这个节点下面创建一个子节点，节点类型是临时顺序节点，也就是说这个节点会随创建者挂掉而挂掉，顺序的意思就是会在节点的名字后面加个数字后缀，唯一标识这个节点在/election的子节点中的id。

一个简单的方案是我们可以每个watcher都watch /election的所有子节点，然后看自己的id是否是最小的，如果是就说明自己是leader，然后告诉应用层自己是leader，让应用层进行后续操作就行了。但是这样会产生惊群效应，因为一个子节点删除，每个watcher都会收到通知，但是至多一个watcher会从follower变为leader。

优化一些的方案是每个节点都关注比自己小一个排位的节点。这样如果id最小的节点挂掉之后，id次小的节点会收到通知然后了解到自己成为了leader，避免了惊群效应。

还有一点需要注意的是，临时顺序节点的临时性体现在一次session而不是一次连接的终止。例如watcher1每次申请节点都叫watcher1，第一次它申请成功的节点全名假设是watcher10002（后面的是zk自动加的序列号），然后下线，watcher10002节点还会存在一段时间，如果这段时间内watcher1再上线，再尝试创建watcher1就会失败，然后之前的节点过一会儿就因为session超时而销毁，这样就相当于这个watcher1消失了。解决方案有两个，可以创建节点前先显式delete一次，也可以通过其他机制保证每次创建节点的名字不同，比如guid。

至于配置下发，就更简单了。配置变更时直接更新节点数据，就能借助zk通知到关注的dbClient，这种事件通知机制相比于轮询请求sentinel要配置数据的机制更加优雅。

我在实现中将zk作为路由协议的一种整合进了Phial规范，这样基于zk的消息通知可以直接走Phial的RPC协议。

有兴趣的同学可以看下我实现的zkAdaptor，leader election的功能作为zkAdaptor的特殊API，watcherService会直接调用。而配置下发直接走了RPC协议，集成在统一的Phial.RPC规范中。zkAdaptor仅支持Phial.RPC中的Notify pattern。

watcher的实现在这里。

5.总结目前形成的架构以及能做什么

整理下这篇文章到目前为止做了什么事情：

- 在文章的一开始确定了游戏服务端要解决的核心两个问题：消息的pipeline与游戏世界状态维护。
- 通过回顾历史的形式提出游戏服务端中最常见的需求情景：多玩家场景同步，并梳理了场景同步最适合的消息pipeline。
- 结合切场景的扩展需求，提出Gate这种基础设施抽象（infrastructure abstraction，简称IA）。
- 尝试进行高内聚、低耦合的服务划分，并总结Gate无法兼顾的消息pipeline。
- 针对Gate无法处理的消息pipeline（service -> service），提出新的MQ-IA，可以大大简化服务间拓扑关系。
- 基于不同的IA与相关协议，提出更高层次的RPC协议，定义了适合.Net2.0和.Net4.5的两种异步RPC调用规范。实现了不同IA到统一规范的Adaptor。总结了游戏中RPC应用的pattern，不同pattern如何与

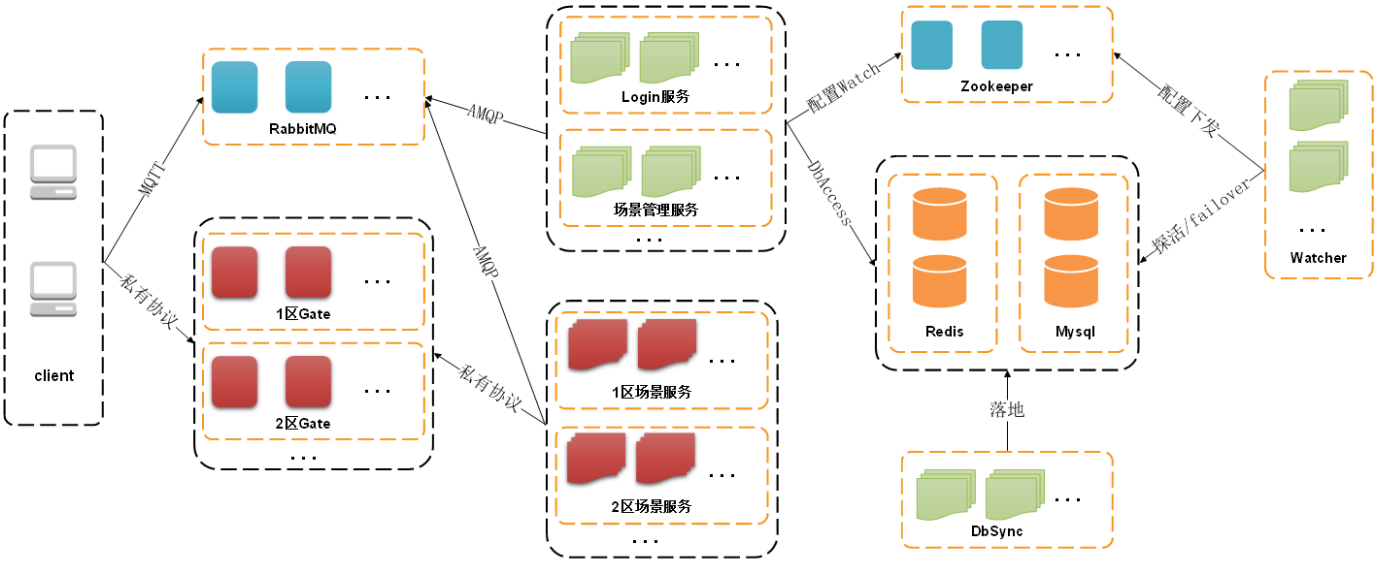
不同IA结合使用。

- 同样通过回顾历史的形式引入数据服务来取代传统MMO中的Db代理进程。
- 结合MQ与数据服务，提出无状态服务在游戏服务端中的应用情景，展开介绍数据服务对于无状态服务的意义所在。
- 基于构建全局数据服务的理念，尝试实现一种多实例的、每实例内向不同服务提供原子修改操作级别一致性的数据服务。
- 为数据服务增加了符合需求的高可用支持。引入了zookeeper，可以让普通的服务也可以复用同样的协调者组件。

总结下出现的几种概念：

- IA。包括Gate、MQ、内存db、持久化强一致性db、分布式协调器等等。不同的IA各司其职，各自只负责解决分布式系统中的一小部分问题。
- RPC与Pattern。面向应用层的统一服务调用方式与规范。
- Adaptor。不同的IA与相关协议到统一RPC与Pattern的适配器。

到目前为止的拓扑图：



系统设计中的static parts与dynamic parts

- static

  gate/mq/zk/redis/mysql

- dynamic

  almost all custom services

这篇文章的灵感起源是the log，看完之后深有感触。虽然JAVA不是一门好语言，但是JAVA技术栈却发展得如此优雅。JAVA技术栈上的每一种IA都专注于解决特定的一小块问题，比如这里提到的。未来的应用框架开发者，就像是用胶水将这些基础设施粘合起来。游戏服务端程序员通常习惯于c++的小圈子，甚至有一种传教的趋势宣扬c++才是代表的游戏服务端的核心技术。有的时候，游戏程序员需要从c++的小圈子跳出来向外走一走，有可能你就不想再湮没在繁文缛节中，而是发现更大的世界。

不过话又说回来，不喜欢跳出c++小圈子的游戏服务端程序员，大部分又都对c++本身其实知之甚少，奉OOP为圭臬，各种虚继承、多继承出来的代码看到想吐。尝试用模板的各种奇技淫巧把c++写成haskell的虽然更有跳出c++小圈子的倾向，但是既然都如此用了，又何必拘泥于c++？

其他

我在这篇文章里尽量少的插入代码，尽量描述游戏服务端定义问题、解决问题的思路。服务端用C#写的毕竟是少数，但是有了思路随便改写成其他语言都没问题。

我顺便也借着写这篇博客的机会，整理了下一些小东西放在github上。

比如之前的面向组合子博客提到的代码生成器组合子，CodeC

比如之前的定时器博客提到的linux内核风格定时器，以及基于定时器写的example，C#协程，都放在这里，CoroutineSharp

比如之前的游戏AI博客提到的行为树编译器原型和c# runtime示例，Behaviour

还有学习parsec的一个小结，可以用来parse单个c#文件拿到一些描述信息的，当然纯属学习性质，有这种需求的时候最好优先用反射。cs_file_parser

然后就是跟这篇博客相关的

一个简单的网络库，Network；

一个简单的基于Network的Gate，GateSharp；

规范的整个底层库，Phial；

为底层库开发的两个配套代码生成器，Phial.CodeGenerator；

示例实现，Phial.Fantasy。

github中的以演示为目的，因此相比于博客，还有不少部分是to be determined（比如详细的配置流程、MQ的集群化、mysql的故障转移集成、落地服务的实现细节等等），之后我也会继续维护。

更多频道

- 小组 - 好的话题、有启发的回复、值得信赖的圈子
- 头条 - 分享和发现有价值的内容与观点
- 相亲 - 为IT单身男女服务的征婚传播平台
- 资源 - 优秀的工具资源导航
- 翻译 - 翻译传播优秀的外文文章
- 文章 - 国内外的精选文章
- 设计 - UI,网页，交互和用户体验
- iOS - 专注iOS技术分享
- 安卓 - 专注Android技术分享
- 前端 - JavaScript, HTML5, CSS
- Java - 专注Java技术分享
- Python - 专注Python技术分享