12306ng应用层子系统服务接口规范建议

12306ng应用层子系统服务接口是指各应用层服务之间，应用层与服务网关层之间的接口。从目前项目组划分来看，应用层包括业务网关、票池、交易网关、SSO以及恶意流量控制系统。为了能让最大优化网络使用效率，减少堵塞，经过票池组对目前已有的一些高并发项目实践的总结，我们向12306ng系统提出应用层子系统服务接口规范建议。

# 服务定义的原则

## 无状态服务

无状态服务是SOA的一个重要原则。使用无状态服务意味着对客户端来说任意两次调用都是不相干的。采用这种原则设计的服务可以有效的减少服务状态管理的成本，提高服务的可扩展性。

## 单向服务（one-way）

单向服务模式中，仅由发送者（caller）向接收者（service）发送一条服务调用消息，接收者不向发送者返回任何消息。换句话说，所有服务提供的方法都是无返回值的，同时服务也不向调用者返回任何确认或错误信息。

如果一个业务操作需要返回值，那么调用方必须实现一个回调服务（callback service）。接受服务计算完成后将调用回调服务将结果通知调用方。

单向调用服务有如下优点：

* 不阻塞调用者当前线程；
* 便于服务将消息队列化处理
* 便于广播消息
* 不依赖编程语言提供的回调模式，减少并发锁的使用
* 消息的发送者与消息返回值处理者可以不必是同一台服务器，更好的支持负载均衡
* 便于将消息持久化处理，以利系统备份/恢复

服务的提供者应当同时提供服务的实现以及服务回调接口的定义；服务调用者自行实现回调服务。服务调用者在设计服务接口时，可以在调用参数里添加Transaction Id 来保持服务调用的上下文关系。

# 服务通信协议

服务通信协议可以分为数据层与传输层两层协议。我们建议数据层协议与传输层协议松散耦合。即传输层可以传输任意数据格式；数据也可以通过不同传输协议传播。（tcp udp）

# 数据层协议

我们提议使用BSON作为统一的数据层协议。BSON的全称是Binary JSON，它是一种二进制编码的类似JSON的持久化格式。如同JSON一样， BSON支持对象或数组中包含子对象或者数组。另外BSON对JSON支持的数据类型进行了扩展，比如日期类型和二进制类型。

相对JSON而言， BSON存储空间更小，解析速度更快，同时还保留了JSON的可扩展性。我们认为比较适合作为服务间调用的标准数据格式。

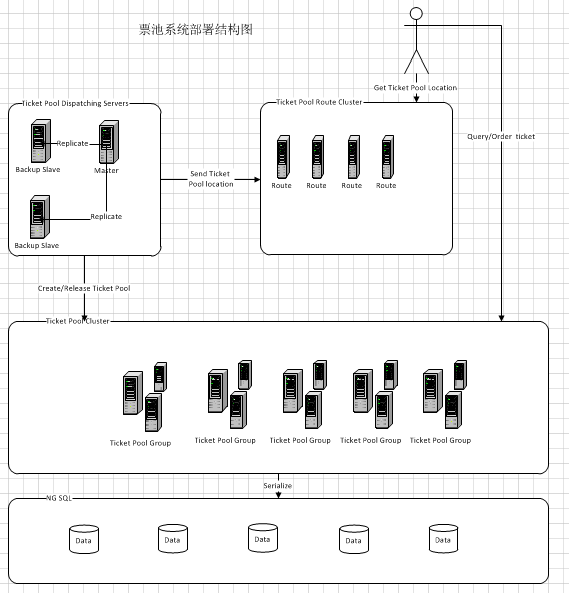
请参考<http://bsonspec.org/>

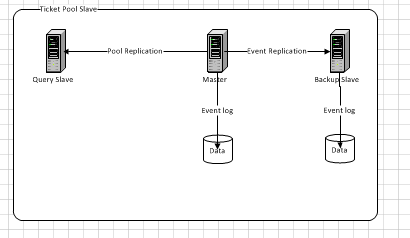
# 传输层协议

在局域网内，除了常用的TCP协议作为传输层协议外，为了提高系统热备份和读写分离的数据同步的性能，我们还建议使用多播协议（multicast）。由于UDP协议不是一种可靠的协议，目前票池开发组正在对一些可靠多播协议(reliable multicast transport protocol)进行调研。

## 票池子系统内的服务架构

我们以票池系统为例说明多播协议在系统中的应用场景。首先解释一下票池子系统的架构模型。





### 票池服务组

票池服务组负责管理1..n运行车次的票池。票池服务组采用类似LMAX Disruptor的架构设计，由三种类型服务器组成：

* 1台Master服务器，负责订购操作并复制信息至slave
* 1..n台 backup slave服务器，负责对系统热备份，当master故障时， backup slave选举其中一台成为master接管工作。每台服务器都已自己独立的event log系统来记录日志。日志的形式可以考虑文件系统以采用stream io提高系统速度。另外还可以实现一个数据交换服务将日志转存至ng SQL或其它应用服务中。
* 0..n 台 query slave 服务器，负责查询操作， master负责保持将票池数据同步到query slave上。

### 票池服务集群

所有的票池服务组的集合，集合中每个票池服务组管理的运行车次不相同。

### 路由服务

负责向用户提供具体运行车次的票池服务的地址，也可以缓存售罄信息。售罄信息可以选择不同的策略，包括整个列车，某个区间，甚至铺位等等。

### 路由服务集群

所有路由服务的集合。其中的每台服务器的路由信息都是相等的。

### 票池调度服务

负责根据负载均衡策略将运行车次的票池分配到具体的票池服务组中，同时通知路由服务运行车次的票池服务地址。当某个运行车次的票池长时间未被使用时，请求调度服务释放此票池。票池调度服务由一个master和多个slave服务器组成。Master负责执行调度服务并将信息复制到slave中去。当master故障时，任何一个slave服务可以选举为master接管调度操作。

### 票池查询订购请求过程

1. 客户端随机选择一台路由服务器
2. 客户端向选定的路由服务发送购票查询信息
3. 路由服务器返回给客户端所查询的运行车次的票池服务组地址

a) 如果路由服务器缓存售罄信息并且用户请求的车票已售罄，直接返回客户售罄消息并退出流程

b) 如果票池服务组支持读写分离，那么路由服务器根据是订购还是查询操作，返回Master或Query Slave服务地址

1. 客户端向票池服务组查询/订购车票

## 使用multicast的传输协议实现系统热备和读写分离数据同步

想法是将查询服务器与车票状态修改服务器分开，通过广播的方式将车票修改状态同步到查询服务器和备份服务器上。

使用广播的传输协议主要是出于节省网络带宽的考虑，因为TCP/IP协议是单点传输协议，同一份数据如果传输到n个查询/备份服务器上的话，就会产生n份重复数据包。但广播协议只需要传输一个数据包到指定的广播地址，由路由负责将数据包传输到加入广播组的服务器上。

然而广播协议，例如UDP协议的问题是，没有容错机制，其不容错主要有以下几个方面：

1. checksum是可选项。
2. 数据包的到达顺序不确定，会因为传输过程中路由的选择，可能序列是1、2的两个包，到达接收服务器的顺序是2、1。
3. 数据包有可能丢失。

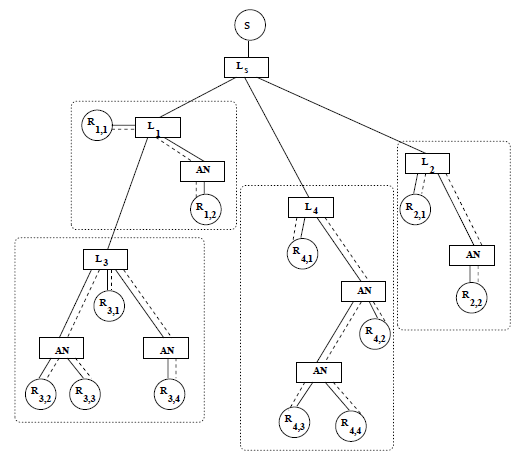
为了解决这个问题，有些人提出了可信广播协议（Reliable Multicast Transport Protocol），其主要就是为了解决前面提到的问题（后两点，第一点通过加上checksum选项解决）。

可信广播协议的基本做法是：

1. 每个数据包都有一个长整形（long – 它的范围已经足够了）的累加序列号，这个序列号一方面是用来做排序，一方面是用来查缺补漏 – 如果接收方发现序列号不连续即知有遗漏。
2. 确认码机制，发送方会等待接收方的确认码，否则会重发数据包。

由于收取确认码可能会造成网络堵塞（多个接收服务器向一个发送服务器发确认码）。在可信广播协议里，确认码机制有以下几种：

1. 确认机制不同：
   1. 正面响应 – 即接收端接收到数据包之后，向发送端返回数据包确认接收。
   2. 负面响应 – 接收端只在发现有遗漏的情况下才向发送端发消息，请求重发遗漏的数据包。
2. 确认码数目的不同：
   1. 不需要确认码。
   2. 只需要一个确认码，即发送端向一批接收端广播，但只要有一个接收端返回确认码即认为广播成功。
   3. 需要k个确认码，即发送端需要收集到n个确认码才认为广播成功。
   4. 所有确认码，即发送端需要收集到所有接收端的确认码才认为广播成功。
3. 确认码传输机制不同：
   1. 直接互联，机器之间不分组，即发送端和接收端在同一个分组里。
   2. 分组互联，即接收端自己组成几个小的分组，每个分组有一个DR负责中转广播并代表分组发送一个确认码给发送端，如下图：



当前找到的可信广播协议有[RMTP](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.4272&rep=rep1&type=pdf)和[PGM](http://www.ietf.org/rfc/rfc3208.txt)协议，开源实现有[jGroups](http://www.jgroups.org)和[OpenPGM](http://code.google.com/p/openpgm/)。对比下来，建议使用jGroups并采用RMTP协议来进行热备和读写分离，主要原因有：

1. PGM协议RFC里就明确说了：

*“PGM has no notion of group membership. ………PGM is not intended for use with applications that depend either upon acknowledged delivery to a known group of recipients, or upon total ordering amongst multiple sources. PGM is best suited to those applications in which members may join and leave at any time, and that are either insensitive to unrecoverable data packet loss or are prepared to resort to application recovery in the event.”*

大意看起来就是不是针对高可靠性和分组设计的。

1. 而jGroups本身就支持分组的概念，从现有的演示视频来看，也正是我们需要的：<http://www.jgroups.org/movies/ReplCache.swf>

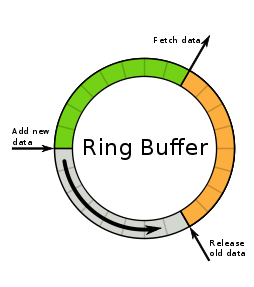
票池广播机制

1. 建议采用分组广播的机制，备份服务器是一组，查询服务器是另外一组。
2. 建议采用正面响应的确认码，在买票时，用户应该可以接受短暂延时，如果采用负面响应的确认码，则写入服务器需要等待一个固定的时间，才能知道是否备份成功，个人觉得这种方式不如正面响应更简单方便。
3. 建议写入服务器到查询服务器的广播可以采取不需要确认码的机制，即使第一次广播的数据有丢失的情况，后面的广播可以修复这个错误。这种机制要求写入服务器到查询服务器的同步是将整个车次的余票信息覆盖同步，而不是增量同步。我认为覆盖同步是可行的，因为：
   * 1. 一开始查询服务器不需要精确到每个座位是否有票，只需要判断车次是否有票即可。
     2. 只有在车次剩余少数几张票的时候，才需要细化到座位是否有票，这个时候，一次广播的数据包依然可以容下这些信息。
     3. 每台写入服务器应该只维护一部分车次的余票信息，车次的余票信息可以分批广播出去。
4. 建议模块之间的确认码使用只需要一个确认码的机制，对于各模块来说，通过假定其他模块内部有合适的容错机制来节省网络流量。
5. 备份服务器可以采用全部确认码或者一半以上确认码的形式，广播机制是增量广播：
   1. 全部确认码机制比较安全，但效率上会慢一些。
   2. 一半以上确认码的机制，这种机制可以看成一个分布式版本控制系统（MVCC），即每台服务器保存备份消息队列的不同消息的分支，但是整个备份服务器分组具有一个完整的版本。例如，写入服务器实际广播了1、2、3个消息，备份服务器A可能只有1、2两个消息，备服B只有1、3，而C有2、3。在写入服务器崩溃时，需要进行版本合并才能形成一个完整版本。
   3. 现在还不确定哪种方案更佳。

## 结合LMAX Disruptor模式实现无锁服务

服务器网关之间的消息传递建议使用disruptor，disruptor的核心理念是：

1. 无锁编程，其使用车轮队列的方式，可以允许多个读线程和一个（甚至是多个）写线程并行访问队列，如下图：



车轮队列的核心思想是：预先分配好内存（足够大，预先分配容量是几百万个元素的队列），避免内存重新分配和GC；读写线程通过CAS（Compare and Swap）操作队列的索引来同步。

1. CPU缓存友好实现，为了避免多核编程中，一个核修改数据，意外导致其他核刷新CPU缓存导致的延迟（伪共享问题），Disruptor通过给对象填充字节来解决这个问题，例如下面两个代码，第一个代码要比第二个代码快上一倍　－　其秘诀就在于第一个代码里为VolatileLong类填充了7个长整形，因为cpu从内存加载数据到缓存是一次64个字节批量读取的，通过显示填充，确保了一个核更新了VolatileLong的数据，不会导致其他核即使没有用到VolatileLong的数据，但是因为缓存里的数据已过期而刷新缓存：

|  |
| --- |
| public final class FalseSharing implements Runnable {  public static int NUM\_THREADS = 4;  public final static long ITERATIONS = 1000L \* 1000L \* 1000L;  private final intarrayIndex;  private static VolatileLong[] longs;  public FalseSharing(final intarrayIndex) {  this.arrayIndex = arrayIndex;  }  public static void main(final String[] args) throws Exception {  System.out.println("开始");  if (args.length == 1) {  NUM\_THREADS = Integer.parseInt(args[0]);  }  longs = new VolatileLong[NUM\_THREADS];  for (inti = 0; i<longs.length; i++) {  longs[i] = new VolatileLong();  }  final long start = System.nanoTime();  runTest();  System.out.println("运行时间 = " + (System.nanoTime() - start));  }  private static void runTest() throws InterruptedException {  Thread[] threads = new Thread[NUM\_THREADS];  for (inti = 0; i<threads.length; i++) {  threads[i] = new Thread(new FalseSharing(i));  }  for (Thread t : threads) {  t.start();  }  for (Thread t : threads) {  t.join();  }  }  public void run() {  long i = ITERATIONS + 1;  while (0 != --i) {  longs[arrayIndex].value = i;  }  }  public final static class VolatileLong {  public volatile long value = 0L;  public long p1, p2, p3, p4, p5, p6, p7;  }  } |

|  |
| --- |
| public final class FalseSharingSlow implements Runnable {  public static int NUM\_THREADS = 4;  public final static long ITERATIONS = 1000L \* 1000L \* 1000L;  private final intarrayIndex;  private static VolatileLong[] longs;  public FalseSharingSlow(final intarrayIndex) {  this.arrayIndex = arrayIndex;  }  public static void main(final String[] args) throws Exception {  System.out.println("开始....");  if (args.length == 1) {  NUM\_THREADS = Integer.parseInt(args[0]);  }  longs = new VolatileLong[NUM\_THREADS];  for (inti = 0; i<longs.length; i++) {  longs[i] = new VolatileLong();  }  final long start = System.nanoTime();  runTest();  System.out.println("运行时间 = " + (System.nanoTime() - start));  }  private static void runTest() throws InterruptedException {  Thread[] threads = new Thread[NUM\_THREADS];  for (inti = 0; i<threads.length; i++) {  threads[i] = new Thread(new FalseSharingSlow(i));  }  for (Thread t : threads) {  t.start();  }  for (Thread t : threads) {  t.join();  }  }  public void run() {  long i = ITERATIONS + 1;  while (0 != --i) {  longs[arrayIndex].value = i;  }  }  public final static class VolatileLong {  public volatile long value = 0L;  }  } |

1. 内存屏障，在多核编程里，因为CPU指令对内存读写指令的优化 – 即CPU保证单核运行时，内存读写的顺序在语义上与源码一致，而实际上出于内存访问速度的优化，会批量读取或者写入一些数据，导致机器指令级别读写的顺序可能与源码不一致。但这种优化在多核编程里，会造成很大问题，例如下面的代码：

Processor #1:

while(f ==0)

;

// Memory fence required here

print x;

Processor #2:

x =42;

// Memory fence required here

f =1;

本来程序的语义是核#2将x设置为42后，在更新f，这样核#1判断f时，退出循环，打印x的值为42。

但是根据CPU的优化，核#2可能会先设置f为1，然后核#1这时执行发现f已经变化退出循环，打印x的值。

为了解决这个问题，disruptor提供了内存屏障的类，确保多核里指令运行顺序和源码一致。