MMServerEngine

## 概述

1. 是什么

MMServerEngine（简称MMSE）是一个游戏服务器引擎。也可用于非游戏的服务端应用，支持如下几点：

1、集群：可设置多个逻辑处理节点。

2、缓存：支持本地和远程两层缓存

3、异步数据库：对于更新，插入和删除操作支持异步更新到数据库

4、服务层事务：任何一个服务处理可以添加事务。

5、房间：用于交互效率要求较高的全缓存服务。

1. 目标

MMServerEngine的设计秉承着两个目标：

1、插件化：即具体使用的技术（如缓存，网络，数据库等）可以通过插件的形式加入到本框架。

2、使用高效：使游戏逻辑编写者尽量不用关注和具体游戏逻辑无关的枝节。

1. 基本框架结构

1、集群框架

mainServer

nodeServer

nodeServer

.......

asyncServer

remoteCache

client

2、Server框架

网络入口（编码解码）

request入口

netEvent入口

request解码

service（业务逻辑）

dataCenter（数据访问服务）

cache服务

async（异步服务）

DB

本地

公共

Aop/Ioc

Monitor

Bean容器

room入口

room解码

roomService

3、Service框架

service/singleService

（有无事务）

request

netEvent

update

event

job

其它

room

## 系统详述

1. 从service说起
2. 概念与功能

* Service类是MMSE对外提供服务的基本单元，将一个普通类添加Service注解（com.mm.engine.framework.control.annotation. Service）即可将其声明为Service。Service注解的结构为：  
  **public** @**interface** Service { String init() **default ""**;// Service被加载后系统执行的初始化方法  
   **int** initPriority() **default** 3; *// 初始化的优先级,越小越先初始化,* String destroy() **default ""**;// 系统关闭时的销毁方法 **boolean** singleService() **default false**; //声明为单实例运行  
  }
* 其中，singleService是指，在整个集群服务器中是否只运行一个服务实例，如果是，其它所有的服务器将通过远程调用访问该实例。
* Service类将在系统启动的时候实例化，然后可通过BeanHelper.getServiceBean(XXX.class)获取其实例。
* Service类在实例化时将进行针对Service的处理，包括初始化，销毁，包括Aop和Ioc，其它组件访问的生成等。

1. 扩展

* 可通过ServiceHelper.getMethodsByAnnotation(Class<? extends Annotation> cls)方法获取所有service中添加某种注解的方法
* FrameBean是一种特殊的Service，FrameBean是指系统中一些功能的默认的实现方式可以由用户重新定义，并通过在配置文件中配置来被系统加载并使用，具体看FrameBean。

1. 插件化思想和FrameBean
2. 插件化思想和思路

* 插件化是本系统的一个重要目标，目前实现插件化的部分不是很多，后面要优化为插件化加载的方式。
* 插件化的目标是，让所有可能被替换的部分，都是可以通过配置的方式替换掉，而不影响其它模块。大的方面，如缓存模块，网络模块等，小的方面，如某个对象的存储。

1. FrameBean

* FrameBean是插件化思想的实施方式，即把可以插件化的部分的入口在配置文件中配置，然后加载进入系统，称为FrameBean。
* FrameBean的配置方式如下：
* frameBean.xxx.type=typeClass  
  frameBean.xxx.class=instanceClass
* 其中xxx为该FrameBean的名字，只是一个标识，typeClass为该FrameBean的接口类型，在系统中的调用时通过它完成的，支持系统提供的Aop和Ioc功能，instanceClass为本FrameBean的实现类。
* FrameBean在系统启动的时候实例化，然后通过BeanHelper. getFrameBean(Class<T> cls);获取。

1. Aop与Ioc
2. Aop的概念与功能

* Aop是指动态的对某个方法的进行改造，在其运行前后添加一定的功能逻辑，在MMSE中，Aop主要是针对Service中的方法。一个Aop功能至少包括切面和切点两个部分构成。
* 切面是指添加Aop的方法，其运行前后要进行的处理逻辑，切点指给那些方法添加切面。
* 切面顺序：定义在切面上面，指对所有切面的执行优先级进行排序。

1. Aop的使用

* 切面的实现：通过对一个类继承抽象类AspectProxy，并声明@Aspect注解来将其定义为切面类。其中，实现两个方法：  
  **public void** before(Object object,Class<?> cls, Method method, Object[] params);  
  **public void** after(Object object,Class<?> cls, Method method, Object[] params, Object result);  
  五个参数分别为：  
  object：对应的Service的实例；cls：对应的Service类（注意，object一般会是Service的代理类的实例，即object.getClass获取的不是cls）；method：所执行的方法；params：方法的参数；result：方法的返回值。
* 切面与切点的关系有三种对应方式，分别对应@Aspect中的三个参数：  
  **public** @**interface** Aspect { String[] mark() **default** {}; Class<? **extends** Annotation>[] annotation() **default** {}; String[] pkg() **default** {};}  
  mark对应后面切点注解AspectMark中的mark，当添加AspectMark的方法，其mark数组中存在对应Aspect中的mark时，将执行该切面，添加该AspectMark的类，其所有的方法都将执行该切面；annotation表示所有添加该注解的方法都将执行该切面，添加该注解的类的所有的方法都将执行该切面；pkg表示，在pkg对应的包下的所有类的所有方法都将执行该切面。
* 注解@AspectMark提供一个参数可用来通过mark数组定义切点：  
  **public** @**interface** AspectMark {  
   String[] mark();  
  }
* 通过在切面类上面添加注解@AspectOrder类声明该切面的优先级，AspectOrder提供一个参数value来定义切面执行优先级：  
  **public** @**interface** AspectOrder {  
   **int** value();  
  }  
  其中value值越小，越优先执行

1. Aop扩展

* BeanHelp提供一个实例化方法：  
  **public static** <T>T newAopInstance(Class<T> cls);  
  通过该实例化方法实例化一个对象，其中的方法将拥有Aop功能。注意：该类必须要在配置文件定义的appPackage中。

1. Ioc功能

* 所有Service，FrameService和Entrance中定义的Service和FrameService的引用，都会由系统自动为其赋值。注意：变量可以定义为私有

1. 五大基本控制组件
2. Request

* request是接收外部访问的基本组件，每个request拥有一个独有的操作码（opcode），由request入口进入的网络访问将根据opcode定位到request，由request对请求进行执行，并返回请求结果。Request入口的配置在配置文件中的entrance.request参数，具体看【系统配置文件】
* 通过对Service中的方法添加@Request注解来生命一个方法为request组件： **public** @**interface** Request { **int**opcode();  
  }
* 对应的方法参数类型和返回类型是确定的：  
  **public** RetPacket xxx(Object clientData, Session session);  
  其中clientData为前端传入的数据，session为前端在后端保存的上下文组件（参考Session和Account）。RetPacket为返回给前端的数据：  
  **public interface** RetPacket {  
  **public int** getOpcode(); // 返回的操作码  
  **public boolean** keepSession(); // 是否保存session  
  **public** Object getRetData(); // 返回给客户端的数据  
  }

1. Event

* Event是事件服务，包括四个部分：事件类型，事件数据，时间抛出，事件接收者。
* 通过给Service中的方法添加注解@EventListener来定义一个监听组件：  
  **public** @**interface** EventListener{  
  **int**event();  
  }  
  参数event代表要监听的事件类型。
* 对应的方法参数类型和返回类型是确定的:  
  **public void** xxx(EventData eventData);  
  其中EventData的结构如下：  
  **public class** EventData {  
  **private intevent**; // 事件类型  
  **private** Object **data**; // 事件数据  
  }

|  |  |
| --- | --- |
| 同步执行事件&不广播 | 异步执行事件&不广播 |
| 同步执行事件&广播 | 异步执行事件&广播 |

* 事件抛出：通过EventService的fireEvent方法抛出事件，包括四种抛出方式：  
  同步执行指：在时间抛出的线程中执行事件，完成后返回  
  广播是指：该事件不仅在本服务器中执行，也在集群中其它服务器中执行

1. NetEvent与RPC

* NetEvent是RPC的实现层，实际使用主要以RPC为主即可
* 网络事件（NetEvent）是对Event的一种扩展，指对集群中其它服务器发出事件。NetEvent的网络入口在配置文件中的入口配置entrance.netEvent来配置，默认使用NetEventNettyEntrance
* 给Service中的方法添加NetEvenListener注解使其成为NetEvent的监听者，每个服务器对每种网络事件只能有一个监听者：  
  **public** @**interface** NetEventListener{  
   **int** netEvent();  
  }  
  netEvent对应监听网络事件的类型
* 对应的方法参数类型和返回类型是确定的:  
  **public** NetEventData xxx(NetEventData netEventData);  
  其中NetEventData和EventData的定义结构一样：  
  **public class** NetEventData {  
   **private intnetEvent**; // 事件类型  
   **private** Object **param**; // 事件数据  
  }

|  |  |
| --- | --- |
| 同步抛出事件&广播 | 异步抛出事件&广播 |
| 同步抛出事件&发给mainServer | 异步抛出事件&发给mainServer |
| 同步抛出事件&发给asyncServer | 异步抛出事件&发给asyncServer |
| 同步抛出事件&发给任一server | 异步抛出事件&发给任一server |

* 事件抛出：NetEvent的抛出通过NetEventService来完成，主要包括如下几种方式：  
  其中，广播是指发送给所有的服务器，发给任一服务器需要提供相应服务器的地址或ServerClient对象；同步发送的发法将返回时间执行的结果：单服务器返回NetEventData，广播返回Map<String,NetEventData>
* 远程调用（RPC）是对NetEvent的一层封装，通过RemoteCallService进行调用，主要的调用方式包括：  
  **public** Object remoteCallSyn(String add,Class cls,String methodName,Object... params);  
  **public** Map<String,Object> broadcastRemoteCallSyn(Class cls, String methodName, Object... params);  
  其中：add是远程服务器的地址，cls是对应Service的类，methodName是对应的方法名，params是对应方法的参数。而对于单个服务器的调用，返回值为Object，广播调用返回一个Map，对应每个服务器返回的值。
* 广播方法是对远程调用的又一层封装，当对一个方法添加@BroadcastRPC注解时，对该方法的调用将广播到其它服务器并使之调用该方法:  
  **public** @**interface** BroadcastRPC {  
  **boolean** async() **default false**; // 是否异步调用  
  }  
  其中，参数async决定远程广播调用是采用同步的方式还是异步的方式。注意：被广播调用该方法的服务器不会再次广播该方法的调用。

1. Updatable

* 更新器，用于做定时更新的服务，可以根据设定的参数进行不同频率和方式的更新。更新器由系统加载并运行，系统期间不会停止。
* 通过给一个Service中的方法添加@Updatable注解来定义一个更新器组件：  
  **public** @**interface** Updatable {  
   **boolean** isAsynchronous() **default true**; **boolean** singleService() **default false**; **int** cycle() **default** -1; String cronExpression() **default ""**;  
  }  
  isAsynchronous：是否异步更新，当设置为false时，该组件进行同步更新，此时cycle和cronExpression两个参数不起作用，在某个固定周期（由系统参数syncUpdate.cycle配置）下，与其它同步更新组件同步运行，所以同步更新服务不应该处理任务量较大的服务。但同步更新不需要再开线程，更不需要cronExpression的低效的时间判断，进而适合做简单的快速更新。  
  singleService：和Serive的singleService一样，当设定为true时，该组件只在一个服务器上面运行。  
  cycle：更新周期，该组件更新的时间间隔，当cronExpression设置有效值后，该参数无效。  
  cronExpression：cronExpression表达式，用于设置更新周期，详见……
* 对应的方法和参数是固定的：  
  **public void** xxx(**int** interval);  
  其中interval是更新间隔，即上次更新到本次更新的时间。

1. Job

* Job是一个定时任务，即在某段时间之后或某个时间点执行一个服务。
* Job的构建和使用：构建一个Job对象，通过JobService的：  
  **public void** startJob(Job job);  
  **public void** deleteJob(String id);  
  来启动和删除Job，其中Job对象的结构为：  
  **public class** Job{  
  **private** String **id**;*// job唯一id***private** Date **startDate**; *// 执行时间***private boolean db**; *// 是否持久化***private** String **method**;  
  **private** Class **serviceClass**;  
  **private** Object[] **para**;  
  }  
  id：集群服务器中唯一id，建议根据业务逻辑和对应的Service同一设定。  
  startDate：job执行时间，如果小于当前时间将立刻执行。  
  db：job是否持久化，即如果服务器重启，是否还要执行该job。  
  method：job执行的方法；serviceClass：job对应的service类；para：job方法接收的参数。注意：如果db为true，para必须是可以序列化的：
* Job的存储：Job的存储方案类的定义需要继承JobStorage接口，并在配置文件中定义jobStorage。系统默认实现了DefaultJobStorage类，并将其存储在数据库中，如果使用它，要求数据库中定义相应的表：  
  **DROP TABLE** IF **EXISTS** `job`;  
  **CREATE TABLE** `job` (  
  **`id` varchar**(255) **NOT NULL**,  
  **`startDate` timestamp NULL DEFAULT NULL**,  
  **`db` int**(11) **DEFAULT NULL**,  
  **`method` varchar**(255) **DEFAULT NULL**,  
  **`serviceClass` varchar**(255) **DEFAULT NULL**,  
  **`params`** blob,  
  **PRIMARY KEY** (**`id`**)  
  ) ENGINE=InnoDB **DEFAULT** CHARSET=utf8;
* 注意：jobStorage属于FrameService，所以支持相应的Aop和Ioc功能

1. 用room做高实时性服务
2. 整体思路

* 为了满足实时性要求较高的功能，如实时对战，实时的场景等，设计了房间模块（com.mm.engine.framework.control.room. Room）。
* 对于实时性要求较高的功能，房间通过两个方面来提高其效率：一是，进入一个房间的玩家需要与房间所在服务器建立socket连接；二是，房间的所有活动可以做全缓存。所以，每个服务器都有一个固定的room的socket入口，由配置文件中的entrance.room配置，具体请看【系统配置文件】部分。对于数据的持久化，可以使用异步存储数据。
* 与request不同，room内的每个命令除了操作码（opcode），还要提供一个房间号（roomId）。在进入房间之前，玩家需要从其它途径获取该房间的roomId（一般为房间对应功能的service）和所在服务器的入口，并建立socket连接（同一个服务器只需要建立一个socket连接，多个room共用，已有的不需要重新建立），然后发送进入房间请求来进入房间。

1. 具体使用

* 具体可以参考游戏案例live。
* 继承抽象类Room，创建自己的房间类，并实现抽象方法，包括：  
  **public abstract void** onInit();  
  **public abstract** RetPacket handle(Session session, **int** opcode, **byte**[] data) **throws** Throwable;  
  **public abstract void** onDestroy();  
  **public abstract void** onPeopleEnterRoom(RoomAccount roomAccount);  
  **public abstract void** onPeopleOutRoom(RoomAccount roomAccount);  
  **public abstract void** onDisconnection(RoomAccount roomAccount);  
  房间的创建是通过RoomService中的方法**public** <T>T createRoom(Class<T> cls);来创建，此时，房间的**void**onInit();方法会被调用，对应的移除房间通过**public** Room removeRoom(**int** id);，并会触发void onDestroy();方法。  
  RetPacket handle(Session session, **int** opcode, **byte**[] data) **throws** Throwable;方法在房间收到消息的时候调用，除了进入房间和退出房间的消息。Room没有添加一层控制来实现对room消息类型opcode针对方法的转发。需要房间类自行判断并处理。（同一个房间往往是针对同一块功能的，没有必要再添加一层控制，但不是不可以考虑）。  
  进入房间、退出房间和网络断线的消息由房间系统统一处理，需要从入口处调用RoomService的**public boolean** enterRoom(**int** roomId,Session session);、**public boolean** outRoom(**int** roomId,Session session);和**public void** netDisconnect(Session session)方法，并会触发自己房间类的**void** onPeopleEnterRoom(RoomAccount roomAccount);、**void** onPeopleOutRoom(RoomAccount roomAccount);和onDisconnection(RoomAccount roomAccount);方法。若使用protocolbuf协议，系统提供的支持protocolbuf协议的Room入口com.mm.engine.sysBean.entrance.RoomNettyPBEntrance支持了这三个方法，具体参考【网络通信】模块。
* 房间模块提供了RoomAccount作为房间中用户的基本数据，包括一个存储数据的map，不过还是建议具体房间功能的类提供新的或继承RoomAccount来作为针对游戏逻辑的数据存储。另外，RoomAccount提供了Session的RoomMessageSender用于推送给客户端消息。
* Room提供了推送广播消息的一些方法，具体参考类源码。

1. 缓存与异步持久化
2. 整体思路

* 为了缓解应用层与持久层交互的效率问题，添加了缓存层，应用层直接与缓存层进行数据交互，并返回，而持久化通过异步的方式由异步模块进行，拓扑图：

service（业务逻辑）

dataCenter（数据访问服务）

cache服务

async（异步服务）

DB

本地

公共

1. ORM的设计和数据库的访问

* 为方便使用存储功能，框架实现了orm（关系对象模型），即每一个数据库表都要对应一个类，进而使得对数据的存储就像直接操作类对象一样。
* 框架选用了mysql来持久化数据，其基本参数需要在配置文件中进行配置。
* 对于与数据库表对应的类，需要满足以下几个要求：  
  第一，要能够序列化，这是因为数据库的数据要能够被缓存。  
  第二，要添加注解@DBEntity：  
  **public** @**interface** DBEntity {  
   String tableName(); *// 表名*String[] pks(); *// 主键*}  
  其中：tableName为对应的表的名称。pks为表的主键，如果为空，将使用表中所有的字段作为主键，但建议不要为空。  
  第三，对于表中字段与类中属性的对应关系，可以通过对勒种属性添加注解@Column来标注：  
  **public** @**interface** Column {  
   String value();  
  }  
  其中value为对应的表中字段名称，此外，如果不添加该注解，系统将根据名称进行匹配，注意大小写敏感。
* DBEntity类中必须包括所有的对应数据库中的字段的属性，反之不需要，即类中可以有对应的其它的属性。
* 对数据库的访问通过静态类DataSet进行，DataSet提供了多种访问方式。主要包括（一部分）：  
  **public static** <T>T select(Class<T> entityClass, String condition, Object... params);  
  **public static** <T> List<T> selectList(Class<T> entityClass, String condition, Object... params);  
  **public static** List<ColumnDesc> getTableDesc(String tableName);  
  **public static boolean** insert(Object entity);  
  **public static boolean** update(Object entityObject,String condition, Object... params);  
  **public static boolean** delete(Class<?> entityClass, String condition, Object... params);  
  其中，entityClass是要操作的DBEntity类，condition为对应的条件，如”id=5”，params是condition中对应的参数，用于替换condition中的”?”。

1. 缓存数据的结构和Key的设计

* 数据的缓存是以key-value的结构进行的存储的，类似于map，其中value用CacheEntity包装了一下：  
  **public class** CacheEntity **implements** Serializable{  
  **private** Object **entity**;  
  **private** CacheEntityState **state**;  
  **private long casUnique**;  
  }  
  其中， **entity**为要存储的数据，需要能够被序列化， **state**为存储数据的状态：  
  **public static enum** CacheEntityState{  
  ***Normal***,  
  ***Delete***, *// 这个说明该数据已经被删除****HasNot****//说明数据库中也没有，这样就不要穿透到数据库判断一个没有的数据，可以考虑用Delete？*}  
  这样，当数据被删除的时候需要做个标记，否则会穿透到数据库进行查询。**casUnique**为本数据的版本号，即所有缓存中的数据都有一个版本号，当数据进行获取-更新操作的时候，通过版本号来防止多线程更新问题。
* 对于对数据库中数据的缓存，其具体数据包括两种类型，一种是对象类型Object，一种是集合类型List，其中，对象类型中存储的是具体的一个数据，即对应数据库中表中的一条记录，而List类型存储的是一系列对象类型的缓存数据的key的列表，并不存储真实的对象数据，对数据的定位要通过二次查询。
* 对象类型的key是由对象的类的类型名称与该对象的主键值组成，以确保其唯一性，组成规则为ClassName\_key1\_key2\_key3\_...。而集合类型的key的组成规则为ClassName#list#条件列名1\_条件列名2\_...#条件值1\_条件值2\_...，其中条件列为查询该结合时使用的查询条件。

1. 两层缓存

* 对数据的缓存分为本地缓存和远程缓存两部分，本地缓存是对远程缓存的补充，即集群中所有服务器公用同一套远程缓存，本地缓存仅对查询的结果进行缓存，以提高再次查询效率。
* 对于查询操作，优先查询本地缓存，没有则查询远程缓存，没有则查询数据库，并填充两级缓存。而增删改要做三件事情：一是，将会清理本地缓存的该数据，并通知其它服务器清理本地缓存的该数据（异步），以确保多服务器之间数据的正常；二是，更新远程缓存；三是，通知异步服务器对数据库进行异步更新。
* 本地缓存使用的是ehcache，远程缓存使用的是memcached，对应客户端为xmemcached，两者可以通过插件化的方式进行配置。

1. 异步更新数据库

* 对数据库的异步更新操作，包括插入，删除和更新，是通过AsyncService完成的，异步更新操作由异步服务器进行，所有服务器的对数据库的操作都将发送给异步服务器处理。
* AsyncService主要对外提供的调用方法包括：  
  **public void** insert(String key,Object entity);  
  **public void** update(String key,Object entity);  
  **public void** delete(String key,Object entity);
* 而数据将会被构建成AsyncData对象并发送给异步服务器。  
  **public static class** AsyncData{  
  **private** String **key**;  
  **private** OperType **operType**;  
  **private** Object **object**;  
  **private int threadNum**;  
  }  
  其中**key**为数据对应的键值，**operType**为更新类型，即插入、删除和更新。**object**为对应的数据，另外，**threadNum**为更新使用线程，由于异步服务器要开启多个线程来处理数据，而对于同一个service访问请求（往往等价于同一个线程），其对数据的操作必须先到先处理，所以用相同**threadNum**来选定其处理使用的异步线程，放入线程队列，确保其时序性。
* 也可以一次提交给异步服务器多个异步处理请求，通过AsyncService的  
  **public void** asyncData(List<AsyncData> asyncDataList);  
  默认所有的异步更新请求将使用与本线程的其它请求（没有则创建）同一个更新线程，所以，参数中的AsyncData不需要为**threadNum**赋值。
* 由于插入和删除也是用的异步进行的，所以对于穿透到数据库的集合查询，需要对异步服务器的符合查询条件的记录进行获取。

1. 基本使用

* 对数据库数据的访问主要通过DataService即可完成，是数据对应用层的最终入口。它即考虑了缓存和异步更新，又考虑了事务问题，但使用起来较为简单，主要使用如下几个函数：  
  **public** <T>T selectObject(Class<T> entityClass, String condition, Object... params);  
  **public** <T> List<T> selectList(Class<T> entityClass, String condition, Object... params);  
  **public boolean** insert(Object object);  
  **public boolean** update(Object object);  
  **public boolean** delete(Object object)；  
  其中entityClass为要访问的数据类， condition为定位条件，允许里面有参数”?”，用params来替换。查询分为两种，查询一个对象和查询一个列表，需要注意的是，查询一个对象要求condition必须为该类型的主键。
* 目前，更新和删除并没有提供针对条件的处理或者批量处理。虽然目前来说已经可以满足所有的需求，为了提高效率和灵活性，有必要提供更多的操作支持。

1. 服务层事务
2. 整体思路

* 事务要满足四个条件，ACID，即原子性，一致性，隔离性和持久性。系统需要再服务层完成这些特性，即服务层事务，实际使用中具体的服务不一定要完全满足ACID，有的可能只需要满足原子性，有的业务逻辑本身不会出现并发问题，而另外一些要完全满足。
* 由于采用的是异步更新数据库，数据库更新之前数据就被投入使用（此时要求缓存已经正常更新），故异步数据库一旦出现问题，数据将通过其它途径存储，并停服处理，而不是回滚数据库更新，所以数据库可以不考虑事务问题。
* 除了持久性使用了异步存储之外，服务层的事务是通过两个服务来完成的，一个是事务缓存服务TxCacheService,一个是锁服务LockerService。TxCacheService可以将事务所涉及的数据缓存起来，直到服务正常完成才提交，否则不提交也便是数据回滚，确保了数据的原子性，LockerService通过对提交数据的加锁来确保数据更新的一致性和隔离性。
* 事务的嵌套采用如下方案：如果在事务之中，则使用之，否则，如果需要事务，则开启事务。

1. 锁服务

* 锁服务运行在一个服务器中，提供集群中所有服务器的加锁服务，目前运行在主服务器上。
* 锁服务对对提供两种加锁方式，一是对某个字符串加锁，二是对一个LockerData进行校验后加锁，LockerData提供的是异步服务器更新数据之前的校验参数，实则也是对字符串加锁：  
  **public static class** LockerData **implements** Serializable,Comparable<LockerData>{  
  **private** String **key**;  
  **private** OperType **operType**;  
  **private long casUnique**;  
  }  
  **key**异步数据的key值，**operType**异步操作类型，**casUnique**缓存数据版本校验。
* LockerService主要提供三个对外的方法：  
  **public boolean** lockAndCheckKeys(LockerData... lockerDatas);  
  **public boolean** lockKeys(String... keys);  
  **public void** unlockKeys(String... keys);  
  **public <T> T** doLockTask(LockTask<T> task, String... keys)；  
  即上面所说的两种加锁方式和一个解锁，还有一种提供一个在锁中执行的任务。
* 锁服务提供的锁为可重入锁，

1. 事务缓存

* 事务缓存相当于给添加事务的服务又添加了一层缓存。对于在事务中的操作，其对单个对象的查询优先从事务缓存中获取，而集合的查询也要从事务中获取新值，而对数据的更新，在事务提交之前只更新事物缓存中的数据，直到事务执行成功，数据统一提交缓存和异步服务器，否则，事务失败，事务缓存中数据抛弃。
* 事务缓存中的数据是通过key-PrepareCachedData的方式存储，其中，key为对应数据的缓存键值，PrepareCachedData为：  
  **public static class** PrepareCachedData {  
  **private** OperType **operType**;  
  **private** String **key**;  
  **private** Object **data**;  
  }  
  **key**异步数据的key值，**operType**异步操作类型，**data**实际缓存的数据
* 事务缓存中记录有本事务需要加锁的类型，用于在事务提交的时候对更新数据进行加锁。
* 事务通过AOP功能实现，对需要添加事务的方法，在方法进行之前进入并初始化事务，方法退出时提交事务。在初始化事务时需要记录需要加锁的类型，提交事务的时候，要先进行加锁校验**public boolean** lockAndCheckKeys(LockerData... lockerDatas)，通过校验后要先提交缓存，最后提交给异步服务器。

1. 基本使用

* 所有的Service中的方法都可以添加事务。需要添加事务的方法，需要添加注解@Tx：  
  **public** @**interface** Tx {**boolean** tx() **default true**;  
  **boolean** lock() **default false**;  
   Class<?>[] lockClass() **default** {};  
  }  
  tx表示是否添加事务，默认是添加的，lock表示事务是否加锁，默认不加锁。lockClass表示需要加锁的数据类型。如果lock为false，lockClass将不起作用，如果lockClass为空，lock为true，将对所有的更新数据进行加锁。  
  事实上，很多服务并不会有并发问题，业务逻辑本身就不会导致并发问题，如对同一个玩家的多个数据进行操作，而这些数据只要改玩家才回修改，但是要保证原子性，所以不用加锁。
* 注意，事务对效率存在较小的影响，加锁影响稍大，尽量减少在同一个事务中加过多的锁，尤其是对于并发要求较高的对象，容易导致锁等待和锁超时。但对于一般服务来讲，在不能确定不加锁就能保证安全的情况下，加锁是没错的。

1. 网络通信
2. 基本概念和思路

* 系统的网络通信是指，通过网络与其它系统与服务进行数据交换，包括作为服务器，接收来自客户端的连接和通信，和作为客户端与其它服务器建立连接和通信。
* 入口（Entrance）是系统作为服务器时的需要构建的组件，一个入口代表一个接受client连接的服务器，是连接通用网络框架，如netty，jetty等和系统应用的组件。入口中需要完成数据包的获取，编解码，对处理服务的调用和数据的返回，最终的异常处理等。系统中必须拥有的入口包括：netEvent入口，即集群中多个服务器之间通信的入口；request入口，即应用客户端与系统数据通信的入口；mainRequest入口，即应用通过mainServer获取它锁登陆的服务器的信息，并进行一个登陆处理；room入口；gm指令入口。
* 系统作为客户端与其它服务器建立连接通过ServerClient进行，即与集群内的服务器的netEvent建立连接，每个服务器均与集群内的其它每一个服务器建立两个client-server连接，互为客户端服务器，所有的数据通信的发起方都是作为客户端完成。

1. 自定义网络通信组件

* 首先，需要先定义一个入口，需要继承抽象类Entrance：  
  **public abstract class** Entrance {  
   **protected** String **name**;  
   **protected int port**;  
   **public** Entrance(){}  
   @AspectMark(mark = {**"EntranceStart"**})  
   **public abstract void** start() **throws** Exception;  
   **public abstract void** stop() **throws** Exception;  
  }  
  其中，**name**为入口名称，**port**为服务端口，start在系统启动的时候被调用，stop在系统关闭的时候被调用。而开始方法添加了aop，是为了在系统启动的时候添加一个事件SysConstantDefine.Event\_EntranceStart，事件参数为对应的Entrance对象。
* 然后在配置文件中配置该入口：  
  **entrance.xxx.port** = **port  
  entrance.xxx.class** = **class**其中xxx为入口的名称，port为服务端口号，class为对应的Entrance类
* 入口提供了类似service的部分系统服务，包括Ioc，Aop。

1. 系统提供的网络服务组件

* NetEventNettyEntrance。用netty实现的集群中服务器之间通信的入口，通过序列化java对象进行数据传输，其中传输的对象为NetEventData，具体参见“五大基本控制组件中的NetEvent和RPC”。在配置文件中配置为：  
  **entrance.netEvent.port** = **8001  
  entrance.netEvent.class** = **com.mm.engine.framework.net.entrance.socket.NetEventNettyEntrance**
* RequestNettyPBEntrance。用netty实现的通过ProtocolBuf通信协议进行通信的Request入口，即与应用客户端进行通信的入口。数据编码方式为：12byte的包头+数据包，其中包头组成为：contentSize(int)-opcode(int)-id(int)。其中contentSize为包内容的大小，opcode为Request中的请求服务类型，参见“五大控制组件中的Request”，id为包的编号，用于客户端标记请求的返回值来对应请求的发送。  
  在配置文件中配置为：  
  **entrance.request.port** = **8003  
  entrance.request.class** = **com.mm.engine.sysBean.entrance.RequestNettyPBEntrance**
* RequestJettyPBEntrance。用嵌入式jetty实现的通过ProtocolBuf通信协议进行通信的mainRequest入口。数据编码方式为：通过http头的参数key=SysConstantDefine.***opcodeKey***传输opcode，通过数据流传输包内容。在配置文件中配置为：  
  **entrance.mainRequest.port** = **8080  
  entrance.mainRequest.class** = **com.mm.engine.sysBean.entrance.RequestJettyPBEntrance**
* RoomNettyPBEntrance。用netty实现的通过ProtocolBuf通信协议进行通信的Room入口。数据编码方式为：16byte的包头+数据包，其中包头组成为：contentSize(int)-opcode(int)-id(int)-roomId(int)。其中roomId为消息发往的房间id，其它三个和RequestNettyPBEntrance一样。  
  因为room的进入和退出房间的消息由入口调用，所以，在RoomNettyPBEntrance中截取了进入房间和退出房间的两个消息，并调用roomService的对应方法：**public boolean** enterRoom(**int** roomId,Session session);、**public boolean** outRoom(**int** roomId,Session session);。（协议定义请参考前后端协议中的“系统提供协议”）在连接断开的时候调用**public void** netDisconnect(Session session)方法。  
  在配置文件中配置为：  
  **entrance.room.port** = **8002  
  entrance.room.class** = **com.mm.engine.sysBean.entrance.RoomNettyPBEntrance**
* GmEntranceJetty。系统提供个gm指令的入口，由嵌入式Jetty实现。系统提供的gm服务是网页形式的服务，具体参考gm模块。在配置文件中配置为：  
  **entrance.gm.port** = **8081  
  entrance.gm.class** = **com.mm.engine.framework.net.entrance.http.GmEntranceJetty**
* Request和mainRequest中需要处理一些和账号登录登出，以及session的一些处理，具体参见“Session与Account”
* NettyServerClient。用netty实现的继承ServerClient接口的类，集群内服务器之间的通信通过它来进行，在netEvent中在系统建立连接的时候和通信的时候实例化：  
  **public interface** ServerClient {  
   **public void** start() **throws** Exception;  
   **public** Object send(Object msg);  
   **public void** sendWithoutReply(Object msg);  
  }

1. 集群建立起来相互连接

集群的相互连接时通过netEvent系统进行的，在集群中的一个节点启动之后，如果不是mainServer，会主动连接mainServer（通过配置文件中的key=mainServer类获取mainServer地址），然后获取其它Server的存在，并与其它server建立连接，此时，mainServer会通知其它server新节点的加入，其它server会与之建立连接，同样，如果再有新的节点加入，mainServer会通知该节点。

1. Session与Account
2. 概念与思路

* Session是客户端与服务器之间建立的一个连接，在系统中主要用于记录应用客户端和服务器建立起来的本次会话的相关数据，包括登陆时间，客户端ip等等。Session是不持久化的，会话结束，Session即被销毁。Account是代表用户的基本对象，用于存储账户的基本信息，其中，accountId是每个标识每个用户唯一性的最终标记，Account需要持久化。
* Session在用户每次登陆的时候建立，并统一管理，同一个Account同一时间只能建立一个Request的Session，即只能在一个客户端登陆，否则，后登陆的会将之前登陆的顶下来。鉴于此，Request的Session是在分配客户端Node服务器的时候创建并在mainServer上面有一份统一管理。
* Room的Session对每个Account来说，集群中每个服务器可以有一个，因为同一个Account可以同时在多个room中，而不同的room可以分布在不同的服务器上。
* 在引擎中提供的Account和对应的AccountSysService主要是解决账户的登陆登出，掉线，推送等问题

1. Session相关

* **public class** Session{  
   **private** String **url**; *// 客户端请求的url* **private final** String **sessionId**; *//* **private** String **accountId**; *//*  
   **private final** String **ip**; *// 客户端本次会话的ip地址*  
   **private final** Date **createTime**; *//session创建的时间，可记录在线时间*  
   **private** Date **lastUpdateTime**; *// 最后一次操作时间，用于计算session是够过期等*  
   **private** MessageSender **messageSender**; *//推送给Request客户端消息的工具*  
   **private** RoomMessageSender **roomMessageSender**; *//推送给Room客户端消息的工具* **private** Map<String,Object> **attrs**; *//session可存储一些属性*  
  }
* **MessageSender**和**RoomMessageSender**是两个接口**，**用于给相应的用户客户端推送消息，这两个值分别需要在对应的session创建的时候赋值，系统提供了两者的基于Netty和ProtocolBuf的实现NettyPBMessageSender和NettyPBRoomMessageSender。对应的使用可参见RequestNettyPBEntrance和RoomNettyPBEntrance
* Session的基本服务由SessionService提供，主要包括session的创建、存储、获取和移除，更多的和Account相关的服务将由AccountSysService提供。

1. 账号的登陆和登出

* 账号的登陆过程的流程图

登陆mainServer（accountId）

y

是否存在Account

n

是否已经登陆

分配一个nodeServer，并通知其建立session

将sessionId和对应的nodeServer地址返回给客户端

y

客户端通过sessionId登陆nodeServer

n

创建account

通知原登陆者被顶下消息

* 账号的登出分为多种情况：第一，玩家主动登出，登出消息发送给mainServer，mainServer通知nodeServer该account的登出，并清理相关数据；第二，掉线，nodeServer先收到socket断线消息，然后通知mainServer，并清理相关数据；第三，被替换掉线，这个在mainServer判断出来，并nodeServer，推送原来登陆玩家在其他地方登陆的消息，清理相关数据；第四，session过期，nodeServer收到session过期消息，并推送给前端session过期的消息，强制其退出，并通知mainServer。
* mainServer保持着所有NodeServer的状态，NodeServerState：  
  **public class** NodeServerState {  
   **private** ServerInfo **serverInfo**;  
   **private int workload**; *// 负载* **private int accountCount**; *// 账户数量* **private** Set<String> **accountIdSet** = **new** HashSet<>();  
  }  
  **serverInfo**为server的信息， **accountIdSet**为server上登陆的玩家。其中，ServerInfo的结构为：  
  **public class** ServerInfo **implements** Serializable {  
   **private** String **host**; *// server地址*  
   **private int netEventPort**; *//netEvent端口*  
   **private int requestPort**; *// request端口*  
   **private int roomPort**; *//room端口*  
   **private int type**; *//服务器类型*  
  }
* 账号的登陆登出的系统服务由AccountSysService完成，系统提供了如下接口，需要系统使用者在账号服务（在本引擎中提供了基于ProtocolBuf协议的AccountService）中调用：  
  **public** LoginSegment loginMain(String id,String url,String ip);  
  **public void** logoutMain(String id);  
  **public void** loginNodeServer(String id,String sessionId);  
  **public void** netDisconnect(String sessionId);  
  如上面的“账号的登陆过程的流程图”中所示，需要先用loginMain登陆mainServer，并获取被分配的nodeServer信息和sessionId：  
  **public class** LoginSegment {  
   **private** String **host**;  
   **private int port**;  
   **private** String **sessionId**;  
   **private** Account **account**;  
  }  
  然后用对应的信息调用loginNodeServer完成nodeServer的登陆。  
  logoutMain为用户主动退出时需要调用的，netDisconnect为应用掉线的时候需要调用的。具体的使用方式可参见AccountService

1. 向应用客户端的消息推送

* 消息推送是指服务器主动向应用客户端发送消息,使用SendMessageService来完成，最终是使用登陆用户的Session中的MessageSender完成的。
* 推送组：指可以指定某些用户为一个推送组，可以一次性推送给该组内所有用户同一个消息，如联盟，派别等。SendMessageService中关于推送组的方法包括：  
  **public void** putIntoGroup(String groupId,String accountId);  
  **public void** removeOutGroup(String groupId,String accountId);  
  **public void** removeGroup(String groupId);  
  推送组的持久化需要继承SendMessageGroupStorage接口，并在系统配置文件中配置FrameBean：**sendMessageGroupStorage**=xxx来保存SendMessageGroup对象。SendMessageGroup对象添加了注解@DBEntity(tableName = **"sendMessageGroup"**,pks = {**"groupId"**})，所以可以保存在数据库中，系统提供了这个功能，可以参见DefaultSendMessageGroupStorage。
* SendMessageService主要的推送方法包括：  
  **public void** sendMessage(String accountId,**int** opcode,**byte**[] data);  
  **public void** broadcastMessage(**int** opcode,**byte**[] data);  
  **public void** sendGroupMessage(String groupId,**int** opcode,**byte**[] data);  
  其中，sendMessage为推送给某个固定用户，broadcastMessage为推动给集群系统中所有用户，sendGroupMessage为推送给某个组内所有的人。

1. 异常与监控
2. 异常

* 系统在运行过程中出问题时，主要是通过抛异常来停止当前线程，异常分为两种，一种是非应用请求中的异常，一种是应用请求中的异常，其中第二种又分为两种，即和应用业务逻辑直接相关，需要返回给前端消息的，一种是不需要返回给前端消息的。其中需要返回给前端消息的，使用ToClientException，其它的，使用MMException。
* ToClientException结构如下：  
  **public class** ToClientException **extends** RuntimeException {  
   **private int errCode** = -10001;*// 异常代号***private int opcode**; *// 前端请求操作码*  
   **private** String **errMsg** = **null**; *// 消息*  
  }  
  **errCode**可以代表不同的异常类型，**opcode**代表前端发起该请求时的操作码， **errMsg**代表传给前端的消息，如弹框阻止该操作的进行等。
* MMException目前只是作为一个异常提醒。

1. 监控思路

监控部分还没有设计好如何做，但其目标是：第一，启动和关闭条件的监控；第二，极端事件的监控；第三，用户可以定制的监控；第四，日常运行的监控，如负载，网络等。

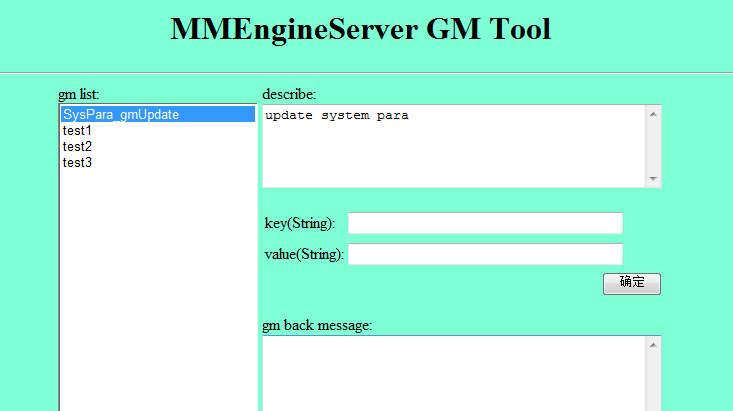
1. 系统工具
2. 系统配置文件

系统配置文件为mmserver.properties，主要包括以下几个部分：frameBean，entrance，mainServer，jdbc（数据库配置），appPacket（系统包，在包内定义的Service才会生效），syncUpdate，用户自定义部分。用户自定义的话，可以通过Server.getEngineConfigure().getString(key);来获取。

1. 系统变量

* 系统变量是指全系统使用的变量，均为key-value类型。有两种设置途径，一个是由配置文件配置，可以由策划配置，另一个是程序自己设置，程序设置的会覆盖掉策划配置的数据。当然，也可以移除程序设置的数据。
* 系统变量服务由SysParaService提供，主要对外提供三个方法：  
  public String get(String key); // 获取一个变量  
  public String put(final String key,final String value); //设置一个变量  
  public String reset(final String key); // 重置一个变量  
  重置变量是指删除程序设置的变量值，如果有配置文件配置的该值，则使用配置文件的值，否则，删除该变量。
* 在MMServerEngine/others/sysPara目录下面是和系统变量的生成和使用相关的工具。在sysPara.properties文件中设置变量，然后运行sysPara.bat，便生成了对应的系统变量源码文件：SysPara.java，默认会将该文件copy到项目的com.sys包下面。如果MMServerEngine/others/sysPara相对于系统的路径修改，请修改sysPara.bat中的copy路径。

1. Gm指令

* 系统提供了方便的gm工具框架。gm工具是通过网页的形式使用的。如果要对一个Service添加一个gm功能，首先编写一个方法，然后给该方法添加一个@Gm注解:  
  **public** @**interface** Gm {String id();  
   String describe() **default ""**;String[] paramsName() **default** {};  
  }id为该gm的唯一id，建议使用Service名+功能，防止重复；describe为该gm工具的描述，在使用界面中会显示；paramsName 为该gm工具的参数的名字，在使用界面中会显示在输入框前面，如果没有则显示“param”。
* GM方法的参数必须是基本类型或者String类型，返回值必须是void或者map<String,String>或者String类型，返回值将在使用界面中作为执行后的反馈.
* gm的入口在配置文件中配置：  
  **entrance.gm.port** = **8081  
  entrance.gm.class** = **com.mm.engine.framework.net.entrance.http.GmEntranceJetty**  
  系统启动的时候日志中会提示使用方式：
* 如在Service中配置gm函数为：  
  @Gm(id=**"SysPara\_gmUpdate"**,describe = **"update system para"**,paramsName = {**"key"**,**"value"**})  
  **public** String xxx(String key,String value){  
   **return "**xxxx**"**;  
  }  
  网页中显示。  
  

1. IdService

* IdService是一个id生成器，可以根据传入的类型Class生成唯一的id，它提供了两个方法：  
  **public int** acquireInt(Class<?> cls);  
  **public void** releaseInt(Class<?> cls, **int** id);  
  acquireInt是获取一个id，releaseInt是释放一个id。
* IdService是属于singleService，生成的id是集群唯一的。

1. 国际化，

还没有做

1. Utils

Utils是系统提供的一些工具，为对org.apache.commons.lang的补充，和一些针对应用的工具，可根据需要添加。

1. 测试

还没有做，基本的思路为：第一，功能测试；第二，压力测试；第三，临界测试。等

## 集群

1. MMServerEngine的集群

MMSE的集群是指多个程序实例同时运行在多个物理机上面，多个实例之间通过网络通信和共用存储空间使得对应用逻辑编写者来讲，只要按照要求的规则编写应用逻辑，就可以将其当做运行在一台物理机上面的一个应用。由于实际上还是运行在多个物理机上面的多个实例，使用的时候没有像运行一个实例那么方便，单一任务运行的效率也会相对较低，但是，通过集群可以更好的分担大量的平行任务，使得集群作为一个整体可以达到单台物理机无法达到的任务负载能力。集群中也会使用很多的方法减少集群导致的应用运行效率的降低，如缓存，异步等。

MMServerEngine集群中有三种服务器实例，主服务器（mainServer），节点服务器（nodeServer）和异步服务器（asyncServer），其中主服务器和异步服务器各自有一个运行实例，节点服务器可以有多个。三者之间的关系参见“基本框架结构-集群框架”

mainServer 的功能：第一，应用客户端的入口，负责给应用客户端分配nodeServer；第二，单服务的运行（目前），包括加载的持久化的job，单服务器运行的update，锁服务，singleService等；第三，负责部分多服务器之间的通信等；第四，所有登陆玩家的登陆状态和所有服务器的状态。等

nodeServer的功能：玩家实际登陆的服务器，主要应用功能逻辑的执行服务器，包括非singleService和room服务。

asyncServer的功能：异步持久化数据的服务器，负责将应用逻辑对数据库的操作异步更新到数据库。

同一个服务器实例可以是多种类型的服务器，如既是mainServer又是nodeServer。

1. 问题及解决方案

不同实例之间的通信是通过netEvent完成的，具体参见“netEvent与RPC”。但通过通信来完成服务器之间沟通会导致多种问题。

1. 数据的可见性问题。

因为每个实例拥有自己的缓存空间，如果将数据存储（或暂时存储）在自身实例中，就可能导致不同实例之间的数据是不可见的。解决方法是，使用共用缓存空间，所有的数据交互都是对共用缓存空间操作，实例自身要么不缓存数据，要么确保缓存的数据与共用缓存空间是同步的。具体参见“缓存与异步持久化”

1. 第二，数据的一致性问题。

在一个实例中，如果有多个线程对同一个数据进行操作，需要通过给各个线程加锁来确保数据的一致性，而在集群中不同实例是无法使用同样的加锁操作。解决方法是，提供一个运行在一个服务器上面的“字符串”加锁服务，集群所有实例可以根据需求来设计统一的字符串作为锁，只有获取该字符串的锁，才能继续相应的数据操作。参考“服务层事务-所服务”。

1. 第三，推送数据的问题。

集群中的不同玩家运行在多个nodeServer上，每个nodeServer只记录登陆自己的玩家状态，并不知道其它玩家的存在，而如果某个功能需要向未在该服务器上登陆的玩家推送消息。解决方法是，mainServer记录了所有的登陆玩家的状态，未在本服务器登陆的玩家的推送交给mainServer，由其交给该玩家登陆的nodeServer推送。

1. 第四，有些服务只能运行一个实例。

singleService是解决单实例运行的方案，当一个服务配置为singleService时，它将只在一个实例上面运行（目前为mainServer），此时可以用作全缓存服务，其它实例都将采用远程调用来完成函数的调用。在非singleService运行的服务器实例上，@EventListener，@NetEventListener，@Updatable和@Gm将不在起作用，@Request将远程调用singleService运行的服务器实例上面的对应方法。

1. 瓶颈

可能导致瓶颈的地方：singleService，asyncServer，locker，cache

## 系统与周边

1. 策划配表
2. 基本使用

策划配数是指策划通过excel设置一些数据，然后在程序中可以作为只读数据调用。MMSE提供的策划配表在MMServerEngine/others/table目录下面。其中tables目录下面是excel文件，然后将table.bat下面的**set *excel\_file\_name***=**xxx.xlsx**设置为对应的excel文件，然后执行之，其数据被生成对应的java文件，在com.table目录下面。

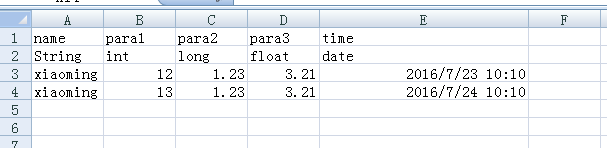
1. excel的配表规则

同一个excel中可以有多个表单，每个表单最多对应一个表，或者不是表。

如果是对应为一个表，那么，表单名字即为表名，即为生成的数据类的类名，第一行为字段名称，第二行为对应字段的数据类型，其与java对象类型的对应关系为"int":"int"，"long":"long"，"String":"String"，"float":"float"，"double":"double"，"date":"java.sql.Timestamp"。从第三行开始为实际的数据。同时该表单中不能有多余的数据，否则，会被认为不是数据表的。

1. 生成的类和使用方式

如果excel的Sheet1配置为



那么生成的类为：**public final class** Sheet1{  
 **public static final** Sheet1[] datas={  
 **new** Sheet1(**"xiaoming"**,(**int**)12.0,(**long**)1.23,(**float**)3.21,java.sql.Timestamp.valueOf(**"2016-7-23 10:10:10"**)),  
 **new** Sheet1(**"xiaoming"**,(**int**)13.0,(**long**)1.23,(**float**)3.21,java.sql.Timestamp.valueOf(**"2016-7-24 10:10:10"**))  
 };  
 **private** String name;  
 **private int** para1;  
 **private long** para2;  
 **private float** para3;  
 **private** java.sql.Timestamp time;  
  
 **public** Sheet1(String name,**int** para1,**long** para2,**float** para3,java.sql.Timestamp time){  
 **this**.name=name;**this**.para1=para1; **this**.para2=para2; **this**.para3=para3; **this**.time=time;  
 }  
 **public** String getName(){**return** name;}  
 **public void** setName(String name){**this**.name=name;}  
 // **其它get set方法**  
}

可以通过静态只读数组变量datas后去所有的配置数据。

1. 前后端协议
2. 使用Protocol Buffer

系统提供了Protocol Buffer协议的使用。在MMServerEngine/others/proto文件夹下面是相关工具。首先将编写的proto文件放在protos文件夹下面，然后打开proto.bat文件，修改里面两个变量，分别是***proto\_file\_name***为proto文件名，***start\_index***为opcode的初始代码。然后运行proto.bat，在com.protocol中即可得到两个java文件，分别是，proto生成的java文件和一个opcode文件，即，该协议文件中的所有的协议的操作码，默认情况下，proto.bat会copy它们到系统的对应目录下面。

在csharp目录下面，有对应的c#代码生成工具，先修改proto.bat文件中的两个变量，分别是***proto\_file\_name***为proto文件名，***start\_index***为opcode的初始代码。然后运行proto.bat，在csharpfile文件夹下生成三个.cs文件，即为对应proto的c#文件。

1. 系统提供协议

* BasePB.proto  
  提供了一个协议，ToClientException返回给客户端的消息
* RoomPB.proto  
  提供了两个协议，进入房间和退出房间的消息。
* AccountPB.proto  
  提供了三个协议：登陆mainServer，登出mainServer以及登陆nodeServer

1. 部署

还没有做

1. 运营

还没有做

## 重要框架原理分析和文件详述

两种类过滤

service代理类的生成：控制组件和aop

bean的创建

服务器事务的实现（提交）

orm的实现

DataService的分析

## 基本使用流程：定义表

1. 先能启动起来

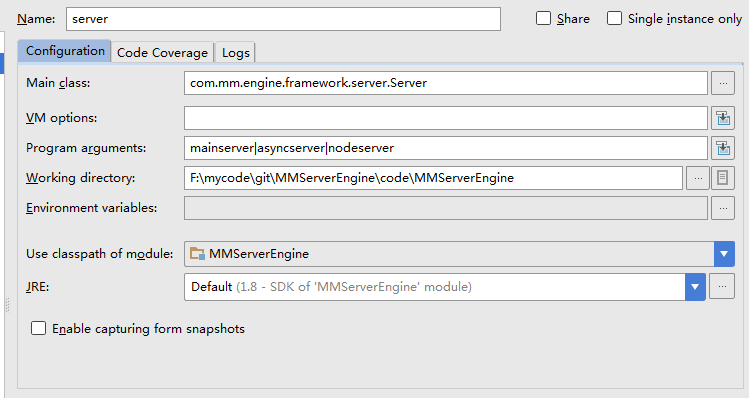
本项目使用windows操作系统IntelliJ IDEA 15.0开发，jdk版本为1.8，建议配置为该环境运行系统，如果使用其它环境，请自行解决兼容问题。

数据库。新建mysql数据库，取名“test”，username=root，password=123456。将MMServerEngine/others/database下面的建表的sql文件在数据库中执行。

memcache。下载windows版本的memcached应用，并启动。

配置文件。第一、mainServer = [自己电脑的ip]:8001，第二，确定jdbc中的mysql参数正确，第三，确保配置文件中的配置的端口号均没有被占用（一般不会，除非运行了其它服务器，如tomcat常用的8080）。

启动参数。打开Run/Debug Configurations对话框，配置一个Application，参数如下：



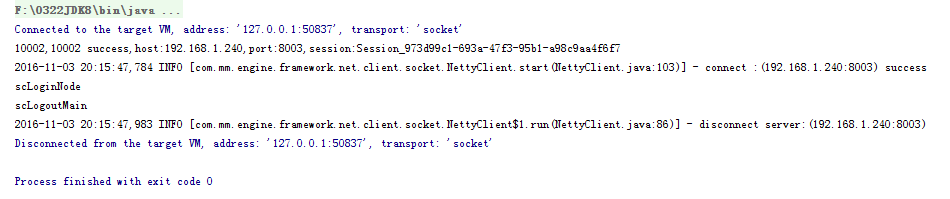
其中，Program arguments说明，该服务实例为三种服务器。

配置完成后运行server，即可启动服务器。如没有异常和error日志，说明正常启动了。

1. 使用系统提供的登陆登出服务

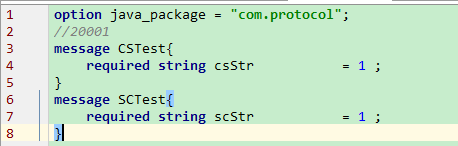
系统提供了基于Protocol Buffer协议的登陆登出功能，参见MMServerEngine/others/proto/protos/AccountPB.proto文件，生成的java文件在com.protocol包中，具体生成方式参见“系统与周边-前后端协议”。

使用的入口为com.mm.engine.sysBean.entrance .RequestNettyPBEntrance，使用的AccountService为com.mm.engine.sysBean.service. AccountService。

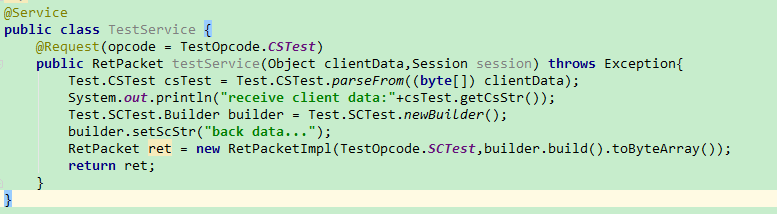
运行com.mm.engine.netTest.AccountTest，其执行过程为：登陆mainServer、登陆nodeServer、登出账号。正常运行结果如下：  


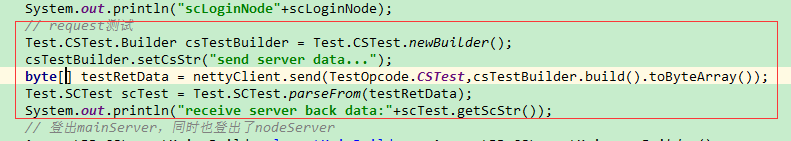
此时数据库中account表有对应的账号数据。

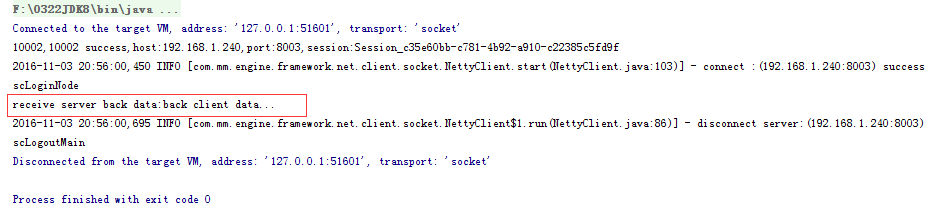
1. 编写自己的Service

在MMServerEngine/others/proto/protos下创建Test.proto，内容为  


运行父目录下面的proto.bat，可以在com.protocol包中得到Test和TestOpcode两个类。

在com.mm.engine.netTest下面编写TestService.java，内容为：  


重新运行服务器，然后，在com.mm.engine.netTest下面创建TestServiceClient.java文件，并将AccountTest的代码copy过来，再在其登陆nodeServer之后添加交互代码：  


运行之，可进行一次服务器运行的交互：  


服务器也将打印相应的日志：  


## 一个小游戏案例Live