## 概述

MMSE

## 系统详述（概念与功能，使用，原理，扩展）

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. 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 int event**; // 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 testNetEvent1(NetEventData netEventData);  
  其中NetEventData和EventData的定义结构一样：  
  **public class** NetEventData {  
   **private int netEvent**; // 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);  
  即上面所说的两种解锁方式和一个加锁

1. 事务缓存（：：需要看看事务嵌套事务开始与结束）

* 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（：：需要主动抛弃吗）。
* 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. Session与Account
3. 异常与监控
4. FrameBean
5. 系统工具：系统变量，Gm指令，IdService，国际化，Utils，测试

## 集群

## 系统与周边

1. 系统配置文件
2. 策划配数
3. 前后端协议
4. 部署
5. 运营

## 重要框架原理分析和文件详述

## 基本使用流程：定义表

## 一个小游戏案例Live