# What Is Zeze

Zeze是一个基于一致性缓存的分布式事务并发应用框架。

Zeze是一个Key-Value数据库。

分布式是Zeze的特性，并发是Zeze的要求。一个分布式并发银行系统有一万个窗口，但所有人都去一个窗口排队，这样的系统显然不能很好的运行。分布式是数据在多台机器间可以安全的共享，并发是对应用的要求-即应用自己的需求需要能并发起来。

“这个世界是天然并发的。”

特性（广告）：

* 数据易访问

访问数据库完全透明。

* 数据完整性严格保护

支持事务。

* 高可用

很容易把服务器设计成无状态的，这样可以把请求分配到任意一台服务器。自然任何一台服务器退出都不会影响系统的可用性。

* 高性能

在缓存命中的情况下，没有任何远程访问。

* 易使用

简单直接的编程接口，直接融入编程语言。支持java&c#

什么样的应用最匹配Zeze

* 应用自然拥有较高的并发。这样便于把负载分布到多个服务器上。
* 访问的数据具有较高的局部性和重复性。这样可以充分利用Zeze的缓存。

比如游戏，当玩家登录后被分配到一台服务器，以后玩家的请求大多数都操作自己的数据。多数时候，不同玩家访问的数据是独立进行，没有交叉，同一个数据会在登录期间，重复被访问。所以游戏是很适合使用Zeze的。

推广一步，一般来说，多数长期在线的应用都适合使用Zeze。

无法充分利用上Zeze的特性，但也可以使用Zeze的系统。比如支付系统。其中个人交易的请求虽然能并发，但请求都是突发的，操作的数据（账户）基本没有不会重复，这样无法充分利用缓存，效率上最终接近后端数据库的性能。支付系统的公司到个人之间的交易能利用上缓存，但由于交易是重要操作，一般会采用马上保存的模式，所以对其中公司账户缓存的利用仅限于读。

# Why Zeze

## 数据修改中途失败怎么处理

如果你对这个问题已经有经验，快速浏览一下下面的分析即可。

如果一个操作需要修改多项数据，但是修改的中途发生异常，那么已经完成修改应该怎么处理？此时的情况非常复杂，有的时候是可以接受的，但多数时候数据已经处于不一致状态，会导致系统出现问题。这种情况的处理方法最合理的方式是放弃所有的修改，恢复到初始状态。就是一般数据库中事务的要嘛全部成功要嘛全部失败的定义。

但事务一般都是数据库支持的，程序的环境很可能没有事务的概念。在没有事务的情况下。一般的解决办法是：修改数据前提前检查全部逻辑条件，最后一起修改。即使这样子，还要祈祷修改过程不要出错。由于一般修改自己的简单变量或者修改已经很可靠的程序语言提供的容器，通常情况下都不会出错，所以这个办法在简单系统中很有效。对于复杂系统，里面有很多模块，模块之间需要互相调用。那么提前检查全部条件就会变得困难。当然也有办法：每个模块提供的每个功能接口分成两部分，一部分是条件检查，一部分是修改。一个业务流程在使用其他模块时，先集中调用所有依赖的条件检查，最后再集中调用所有修改。这种方法看起来很麻烦，很显然已经破坏了业务流程的直观性：即因为集中条件检查这个原因，导致实现代码和实际业务逻辑需求描述不一致。如果被调用的模块又要调用其他模块，会使得情况变得更复杂，分成两个部分的办法还不能解决代码复用（模块依赖）的问题，还需要对整体的代码分布进行考量，把代码分成条件检查和修改层，业务逻辑层（业务逻辑层不能相互依赖）。

有些程序解决中途修改失败的问题的解决办法是先把修改保存在局部变量中最后一起提交，或者先复制一个拷贝在中途失败时恢复。这种做法实际上很像数据库的事务了，只是都是专用的，没有通用的容易使用的系统。Zeze就试图提供这样一个通用的支持事务系统：直接在程序开发时，拥有事务的特性。

为程序开发直接提供事务是Zeze最初的目的。让实现代码和业务描述尽可能近，几乎能对照起来。业务按什么顺序描述逻辑条件，不会因为“提前检查”这个做法破坏调用顺序。模块划分的时候也很自然，不用分两个方法，分成两层，模块嵌套调用也很自然。不管中间什么时候出错，事务保证了所有的修改回滚回初始状态。

## 一个游戏里面得到经验升级的代码例子(c#)。

1. Void TryLevelUp(long NewExperience)
2. {
3. Role.Experience += NewExperience; // 累加上新的经验
4. // 根据经验配置表，判断是否达到升级需要的经验。
5. // 增加一次经验，可能升多级。
6. While (Role.Experience >= NextLevelExperienceConfig[Role.Level])
7. {
8. Role.Experience -= NextLevelExperienceConfig[Level];
9. Role.Level += 1;
10. If (Role.Level % 10 == 0)
11. {
12. // 每10级学到一个新技能。
13. Role.Skills.Add(LearnSkillConfig[Level / 10]);
14. }
15. }
16. }

如果TryLevelUp要求增加一次经验的所有升级是一个事务，而且第13行又可能抛出异常，那么上面的代码在没有事务时就不好处理，有了事务就很直观。

# Quick Start

## 定义数据结构

<?xml version="1.0" encoding="utf-8"?>

<solution name="QuickStart" ModuleIdAllowRanges="1-100,101">

<module name="Role" id="1">

<bean name="BRole">

<variable id="1" name="Level" type="int"/>

<variable id="2" name="Experience" type="long"/>

</bean>

<bean name=”BBag“>

<variable id="1" name="Items" type=”list[int]”/>

</bean>

<table name=”tRole” key=”long” value=”BRole”/> key is roleid

<table name=”tBag” key =”long” value=”BBag”/> key is roleid

<bean name=”BAddExperience”>

<variable id="1" name="Experience" type="long"/>

</bean>

客户端直接增加经验是不合理的，但这个例子就是这样做了，不管作弊啦。

<rpc name=”AddExperience” argument=”BAddExperience”

TransactionLevel=“Serializable“ handle=”server”/>

</module>

<project name="GameServer" scriptdir="src" platform="java">

<service name="Server" handle="server”>

<module ref="Role"/>

</service>

</project>

</solution>

## 生成代码

把上面的xml保存为solution.xml，切换到文件所在目录，执行Gen.exe solution.xml。在当前目录下会创建GameServer目录(project name)，在里面会找到生成的代码。这里先不细说了。

## 使用例子 (java)

// Gen.exe 会根据定义自动生成空的处理函数。里面的代码就是自己的实现了。

@Override

long ProcessAddExperience(QuickStart.Role.AddExperience r) {

var session = ProviderUserSession.get(r); // 这是个魔法，反正拿到会话了。

var roleId = session.getRoleId();

long newExperience = r.Argument.getExperience();

var role = \_tRole.getOrAdd(roleId);

role. setExperience(role.get Experience() + newExperience);

while (role.getExperience() >= ExperienceConfig.get(role.getLevel())) {

role.setExperience(role.get Experience() - ExperienceConfig.get(role.getLevel()));

role.setLevel(role.getLevel() + 1);

if (role.getLevel() % 10 == 0) {

addItemToBag(roleId, LevelRewardItemConfig.get(role.getLevel()));

}

}

return 0;

}

// 多数时候，Bag是另一个模块(module)。这里都定义到一起了。

void addItemToBag(long roleid, int itemId) {

var bag = \_tBag.getOrAdd(roleId);

if (bag.getItems().size() > 100)

throw new RuntimeException(“Bag Is Full”); // 别担心，包满了，所有修改都会回滚。

bag.getItems().add(itemId);

}

## 上手准备

### Quick Quick

1. git clone https://gitee.com/e2wugui/zeze
2. 运行 zeze\gen\_use\_publish.bat。
3. 再次 Build All，应该就正常了。
4. Csharp版使用vs2022或其他兼容编辑器打开zeze/Zeze.sln
5. Java版使用idea打开 zeze/ZezeJava即可正常Build All。

### Java开发准备 (如果只关注C#开发,可跳过)

1. JDK 17 任意发行版均可

推荐: https://adoptium.net/releases

备选: https://jdk.java.net/

2. IntelliJ IDEA 免费社区版(Community)即可, 2021.2.1版以上

https://www.jetbrains.com/idea/download/

3. (可选) Maven: https://maven.apache.org/download.cgi

4. (可选) Ant: https://ant.apache.org/bindownload.cgi

### C#开发准备 (如果只关注Java开发,可只安装.NET SDK)

1. .NET 6 SDK (如果安装下面的Visual Studio,可以不用单独安装这个)

https://dotnet.microsoft.com/en-us/download/dotnet/6.0 (通常选择Windows, x64)

2. Visual Studio 2022 (可用免费社区版,但需要联网激活,且只能个人或小规模商用)

Windows版本: https://visualstudio.microsoft.com/zh-hans/downloads/ (必选组件: .NET桌面开发; 可选组件: Node.js开发)

Mac版本: https://visualstudio.microsoft.com/zh-hans/vs/mac/

3. (可选) VSCode (完全免费的轻量级IDE)

https://code.visualstudio.com/ (安装官方C#插件)

### Java编译

1. 启动IDEA, 打开zeze框架中的 ZezeJava 目录, 会自动加载4个模块(框架核心+测试+2个示例)

2. 如果编译找不到某些类, 需要执行下面的"生成代码"

### C#编译

1. 仅用SDK: 在zeze框架根目录下使用命令行编译: dotnet build Zeze.sln

2. 使用VS2022: 启动VS2022, 打开zeze框架根目录下的Zeze.sln, 执行编译命令

3. 如果编译找不到某些类, 需要执行下面的"生成代码"

### Java示例工程生成代码

需要先执行上面的C#编译Gen工程

1. 在 ZezeJava/ZezexJava, ZezeJava/ZezeJavaTest 目录下执行 gen.bat

### C#示例工程生成代码

需要先执行上面的C#编译Gen工程

1. 在 Sample, UnitTest 目录下执行 gen.bat

# Arch

## 分布式架构



上图是Zeze默认的框架结构，具有一定通用性。某些情况下，自己可以搭建全新的架构。

* Linkd 连接进程，负责负载分配。Zeze提供一个默认实现，配置以后，生成代码，即可零开发直接使用。生成代码的目的是为了支持重载某些方法，实现定制。
* GameServer 主服务器，实现业务逻辑。开发的主要产出。
* ServiceManager 服务注册和发现服务器。Zeze提供，不需要开发。
* GlobalCacheManager 一致性缓存支持服务器。Zeze提供，不需要开发。
* Database 后端数据库，支持Mysql,Sqlserver,Tikv。将来Zeze会根据需要支持更多的数据库。

## 单实例架构

系统内只有一个GameServer，客户端直接连接GameServer。此时不需要ServiceManager和GlobalCacheManager。由于分布式架构本质上也包含了单实例架构，所以这个模式的例子被删除了（可用从git的历史中找到）。需要说明的是，不管那种架构，业务开发代码几乎一样，架构应该仅影响程序框架。如果需要实现全新的架构，最好不要影响业务开发，方便需要的时候进行迁移。

# Transaction

## 存储过程

事务的执行单位是存储过程。通常情况下，应用不需要创建自己的存储过程，可以通过配置或注解声明事务级别（TransactionLevel），然后由Zeze自动创建。

## @TransactionLevel

在定义协议的时候，配置TransactionLevel属性（默认是Serializable，表示需要事务）。

在处理函数前面加注解。注解的优先级最高，会覆盖定义协议时的配置。例子：

@Zeze.Util.TransactionLevelAnnotation(Level=Zeze.Transaction.TransactionLevel.None)

public enum TransactionLevel {

None, // 不需要事务

Serializable, // 可串行化的事务。【Default】

AllowDirtyWhenAllRead, // 当事务没有写操作时，允许脏读。

}

两个账户初始为0，系统并发随机转账（允许结果为负数）。一个统计事务把两个账户加起来得到Sum。当Serializable时，Sum总是为0。当AllowDirtyWhenAllRead时，Sum可能不为0。

### 1. 程序默认TransactionLevel为Serializable

优先级最低。

### 2. Module.DefaultTransactionLevel

配置模块里面协议的默认事务级别。仅应用于当前模块，不包括子模块。模块默认事务级别一开始要计划好，设定了之后就不能改了。

### 3. Protocol.TransactionLevel

协议里面配置这个协议处理时的事务级别。

### 4. @TransactionLevel

协议处理函数前面加这个注解，指定处理函数的事务级别。

### TrasactionLevel的建议配置方式

上面1-4，四个地方可以配置事务级别，优先级从低到高，高优先级如果配置了会覆盖低优先级配置。

1. 模块内大部分协议处理需要Serializable级别的事务时，除了个别用注解或协议级别配置覆盖，其他都不需要配置。
2. 模块内大部分协议处理不需要事务时，配置Module.DefaultTransactionLevel为None，个别需要的用注解或协议级别配置。

## 嵌套存储过程

当业务需要忽略部分失败，并继续执行事务时，就需要嵌套存储过程。此时需要主动创建存储过程。创建存储过程接口为：Zeze.Application.NewProcedure。例子如下：

protected long ProcessMainTransaction(SomeProtocol p) {

// 一些处理

if (0 != App.Zeze.NewProcedure(MyNestProcedure, “MyNestProcedure”).Call()) {

// 一些嵌套存储过程失败的处理，此时MyNestProcedure的修改全部被回滚。

}

// 继续处理

}

private long MyNestProcedure() {

if (someCondition)

return 0; // success

return ErrorCode(1); // fail

}

## Table

Table是存储过程访问数据的接口。

Table的数据结构在Solution.xml中描述。

Table就像一个Map，主要包含的方法：GetOrAdd，Get，Put，Remove。

## 乐观锁 & 不会死锁

Zeze采用乐观锁，事务执行过程中不会对数据加锁，在最后提交时才加锁并检查冲突，如果冲突了就重做事务。

## WhileCommit & WhileRollback

由于事务会重做，即事务内的所有代码都会可能被重复执行。当在事务内发送协议时，重做导致协议可能被发送多次。

* WhileCommit 事务成功提交时执行。
* WhileRollback 事务失败回滚时执行。

这两个方法定义在Zeze.Transaction.Transaction中。

public void VerifyAccountSum() {

var account1 = tableAccount.get(“tom”);

var account2 = talbeAccount.get(“jack”);

var sum = account1.value + account2.value;

Transacton.WhileCommit(() => assert sum == 0);

// 如果没有WhileCommit，即使在TransactionLevel为Serializable，

// 这个断言也会失败。因为乐观锁执行的过程中是不加锁的。

}

## 自定义日志

可以通过Transaction.GetLog、PutLog实现自定义日志完成一些事务相关特殊操作。自定义日志实现的Commit必须成功，否则程序会终止运行。

## 存储过程返回值

* =0: 成功
* <0: Zeze内部错误码
* >0: 用户自定义错误码。编码：(Module.Id << 16) | Module.ErrorCode。Module的基类有个辅助函数Zeze.IModule.ErrorCode(int code)用来构造这种错误码。模块级别的错误码可以在模块定义(solutions.xml)中用枚举(enum)定义。

## 日志

Zeze记录了几乎所有的错误（异常）日志，只要出现问题，查看日志是个好办法。应用实现时，记录自己逻辑相关的日志即可。

## 自动发送Rpc错误结果

Arch框架在处理Rpc的存储过程时，如果得到非零的返回值，会自动发送Rpc的错误结果。异常的时候也会返回错误码。所以一般存储过程的处理流程只需要在正常的时候设置自定义rpc的正常结果参数并调用rpc.SendResult()；错误的时候直接返回错误码。

## 分布式事务

在分布式架构中，Server有多台实例，每一台的代码是一样的。开发Server时，用户可以简单的认为自己单独拥有后台Database的所有数据。在使用事务读写由Zeze-Table管理的数据时，不需要任何额外的操作。数据在多台Server实例间的共享以及一致性保证由Zeze处理。这就相当于任何Server实例上的事务都是分布的了。

## 事务重做导致的问题例子

事务在执行的过程中，如果发生了冲突，会自动重做。应用开发的时候必须注意这个特性，否则会出现麻烦。例子如下：

* 事务中发送协议。除非不在乎协议的重复发送，可以使用直接发送协议的函数，否则应该使用SendWhileCommit,SendWhileRollback等事务相关的版本。
* 事务中注册Timer。使用Task.schedule或者自己的Timer管理器注册Timer时，请使用Transaction.whileCommit,whileRollback注册action完成注册操作。
* 事务中提交任务（Task）给线程池。一般也需要使用whileCommit,whileRollback。
* ***事务中需要操作非Zeze管理的数据（自定义数据）***。上面的Timer，Task实际上也是操作非Zeze管理数据的一种。操作自定义数据，一般都需要使用whileCommit，whileRollback。

## @DispatchMode

这是一个协议处理函数的注解，用来控制协议的调度方式（仅用于Java）：

1. 在普通线程池中执行。默认是这种。
2. 在重要线程池中执行。
3. 在调用者线程执行。

协议的线程调度方式除了用这个注解单独控制。还可以在Zeze.Net.Service子类中重载DispatchProtocol,DispatchRpcResponse等控制。重载会覆盖默认实现，优先级比注解高。默认实现按注解方式调度协议的执行。

# 事务和第三方系统交互问题

## 问题分析

当事务访问的数据全部都是zeze管理的，这种情况下，zeze已经处理了所有问题（几乎吧），实现逻辑时，可以按自己最舒服的方式进行编写逻辑实现。当事务需要操作非zeze管理的数据（这里的数据包括不是zeze管理的内存变量）时，如上一章提到的注册Timer，向ThreadPool提交任务等，这些操作非常可靠，几乎可以认为不会失败，此时可以简单的用Transaction.WhileCommit处理。当事务操作的是第三方系统的数据，由于第三方系统是有可能失败的，此时不能简单的使用WhileCommit处理。如果第三方系统支持两段式提交等事务方式，可以采用并融入本地事务，如果第三方系统不支持事务，需要具体问题具体分析。下面先举个例子。

Transaction(InputParam)

{

Foreach (var item in InputParam.Items)

{

If (false == Bag.Remove(item))

Return false; // rollback

}

// Oss：某个云对象存储系统。

If (false == oss.putObject(InputParam))

Return false; // 希望正确rollback

Return true; // 事务提交

}

上面的流程看起来非常合理，但是已知存在如下问题：

1. 事务可能重做，但最终成功，oss.putObject会写入多次，这不是个大问题，仅仅影响性能。
2. 由于事务可能重做，并且再次尝试时回滚了，但是oss.putObject的数据已经写到第三方系统了，这会导致错误的写入Oss。

## 解决思路

### 还是WhileCommit

如果认为Oss系统足够可靠，那么简单的使用WhileCommit(() => oss.putObject())即可。这种方案不推荐采用，因为这样的第三方系统应该是不存在的。

### 队列啊队列

当第三方系统不支持事务方式操作时，可以引入一个“支持事务的队列”。zeze事务处理时，先写入这个队列，落地持久化。然后这个队列有个订阅者搬运工，把提交的内容搬到真正的第三方系统中。这个“支持事务的队列”的实现，可以千奇百怪了，下面举几个例子。

#### 使用Zeze的Table

使用zeze的table存储将要保存到oss的数据。这就把原来事务内对oss的操作转换成受zeze管理的数据的操作。然后有个搬运线程，不断的把最新提交的数据搬到oss。TODO 抽象并包装成zeze-one-object-queue。

#### 使用本地文件系统

使用本地文件系统存储将要保存到oss的数据。实际上这个就是自己实现一个队列了。注意，保存到文件系统的数据有先后关系，别忘了这是一个队列。即，最新事务提交的数据应该后处理，最终保存到oss的数据是最新的。这个方案的缺点是，不支持分布式。当存在多个server时， server之间的队列之间没有通讯，会导致fifo失效。

#### 使用RocksMQ

对于大系统，可以考虑采用现在已经很成熟的第三方队列。Zeze已经对RocksMQ进行了简单包装。这个方案的缺点是系统复杂度大大提升，需要自己权衡得失。TODO 确认zeze对RocksMQ的包装处理了所有问题。

# Listener

Zeze支持在Table中注册数据变更监听器，当数据发生变化时，回调监听器。不管什么修改操作，不管有多少修改操作，只要对数据进行了修改，都能得到通知。只要数据定义没有变化，修改操作随意变化，监听逻辑都保持不变。这个功能非常一般用于同步数据给客户端。除非逻辑接近事件这种模型，否则不要用监听器实现逻辑。

【TODO】监听数据变化时得到是数据的增量变化，可以实现变更最小通知。由于增量变化的系列化没有完全实现，所以现在还不能无缝支持。

## 分布式？

分布式情况下，Server有多台实例。此时Listener在每个Server上都注册。以后哪台Server发生了修改，就回调哪一台上的Listener。Listener这个模式不能算是一个良好的分布式定义，需要注意。

## 接口

public interface ChangeListener {

void OnChanged(Object key, Changes.Record r);

}

## 例子

public static class ItemsChangeListener implelents ChangeListener {

void OnChanged(Object key, Changes.Record r) {

switch (r.getState()) {

case Changes.Record.Put:

// 记录整个被替换。

break;

case Changes.Record.Edit:

// 增量变化，

var notemap2 = (LogMap2<Integer, BItem>)

c.getVariableLog(tequip.VAR\_Items);

if (null != notemap2) {

// 访问详细日志。对于这里的LogMap2，note里面包含

// Replaced(被替换的项)，

// Removed（被删除的项），

// Changed（Map中的项是一个Bean并且发生了变化）

// 由于同步Map的数据时，Changed通常可以合并到Replaced中。

// 所以首先调用下面的方法合并。

notemap2.MergeChangedToReplaced();

// 然后把Replaced，Removed打包到协议中发送给客户端。

}

break;

case Changes.Record.Remove:

// 记录删除

break;

}

}

}

## 注册

\_tequip.getChangeListenerMap().AddListener(new ItemsChangeListener());

## 客户端收到数据变化协议的处理伪码

Switch (ItemsChangeNotify.getChangeTag()) {

Case Put:

Localmap.clear();

Localmap.putAll(ItemsChangeNotify.Replaced());

// Put时Removed没有意义。

Break;

Case Edit:

Localmap.putAll(ItemsChangeNotify.Replaced());

Localmap.removeAll(ItemsChangeNotify.Removed());

Break;

Case Remove:

Localmap = null;

Break;

}

## 变更日志

1. Bean的日志LogBean，包含IntHashMap<Log> Variables
2. List1的日志LogList1，包含List的操作日志。
3. List2的日志LogList2，包含List的操作日志以及List中的项修改的Changed。
4. Map1的日志LogMap1，包含Replaced，Removed。
5. Map2的日志LogMap2，包含Replaced，Removed，Changed。
6. Set的日志LogSet1，包含Added，Removed。
7. 简单类型（如int）的日志，包含新的值Value。

## 数据同步模式探讨

### 探讨的问题

1. 客户端数据怎么从服务器获取和同步？
2. 客户端获取数据的时机？
3. 客户端数据存储（生命期）探讨？

### 数据获取（侧重和服务器交互）

#### Get 模式

客户端主动发送Get请求，服务器返回结果。

【这个模式大大的推荐！】

#### Push模式

客户端缓存数据，并且通过交互协议与服务器数据进行同步，客户端数据总是最新的。一般用于登录期间一直有效的数据。客户端初始Get一次或者0 Get或者订阅方案。服务器主动通告最新数据给客户端。

##### Push 完整数据

当服务器发现数据变更时，把最新数据打包Push给客户端。客户端数据结构和服务器数据结构不一致，打包时可做翻译。这个模式结构清晰，实现简单。

【这个模式大大的推荐！】

##### Push 不完备增量更新

客户端主动Get一次数据，以后服务器Push增量更新给客户端。这里不完备的意思是仅打包部分修改Log（比如Bean的第一级变量的Map，Set类型相关的修改数据）。

【这个模式限制使用！迫不得已时采用！】

这个模式有两个问题：a）Get和增量更新原子问题。b）增量更新消息丢失。这两个问题的推荐解决方案：在数据中记录一个修改版本号，修改时递增；Push时带上版本号；客户端发现版本号出现错误，重新Get再次开启新的增量流程。

##### Push 完备增量更新

完备的意思是任何Bean结构的增量更新。在客户端实现这个需要的代码较多。目前只有conf+cs+net支持（还没测试过）。

【这个模式暂时不推荐使用！】

##### Zeze.Listener 的使用场景（限制）

Listener适合同步个人数据。比如角色包裹，服务器监听Listener，发现改变发送push-notify给该包裹拥有者，客户端接收push-notify并更新本地数据。Listener不适合更新共享的数据，比如群成员列表这个数据的变化，虽然可以发notify广播，但有点浪费，【实在要使用也可以考虑这样做】。

### 客户端数据获取时机

#### 在Loading界面准备数据，然后一起显示。

#### 先显示UI，然后获取数据，再逐渐显示出来。

#### 客户端时机点

1. Auth 验证
2. Login 登录
3. Map.EnterWorld 进入地图场景
4. UI显示时
5. 其他需要数据的时候（对于服务器Auth是必须的）
6. 客户端数据存储管理模式

### 客户端数据存储管理模式

#### 全局数据管理器

不够灵活，不太容易满足各种需求。

#### 模块自己管理

个人建议这一种。当模块需要热拔插时尤其需要这种。

# Solution.xml

## Solution

解决方案。一个应用系统可以包含多个解决方案。配置文件名字可以不叫solution.xml。但存在多个解决方案配置文件时，建议按solution.xxx.xml方式命名文件。

## Import

引入其他solution配置文件，这样就可以使用其他文件内定义的Bean。两个soltion配置文件可以循环Import。如：<import file="solution.linkd.xml"/>

## Module

模块在Zeze里是定义bean，table，protocol，rpc等的地方。这个名字和系统功能划分的“模块“的概念一致。Zeze本身没有为模块提供什么重要功能，在生成代码时，主要提供内部对象的名字空间。module可以包含module。module.id必须唯一，必须在solution. ModuleIdAllowRanges规定的范围内。当系统包含多个solution时，范围不能重叠。最终保证了module.id在整个系统内唯一。生成代码时，除了生成模块内定义的对象，还会为每个模块生成一个入口文件。需要处理的协议会在这个入口文件内生成空的处理函数。

## Bean

Bean是Zeze的核心对象，用来定义数据结构。其中的variable描述变量名字和类型。variable可以自由增删变量（即使系统上线以后），自动兼容旧的数据结构。variable.id在bean内唯一，不能复用（variable删除以后，新增的变量再次使用相同的id被认为是一个反悔操作，此时variable.type必须和删除前的兼容）。variable.type可以是另一个bean。Bean的命名建议以“B“开头。这样需要创建Bean时，输入B即可得到编辑器的提示。

## Table

定义Key-Value存储表。

<table name=”tTrade“ key=”long” value=”BTradesession” />

## Protocol

定义协议。

<protocol name=”Trade” argument=”BTrade” handle=”server”/>

argument是一个Bean。

handle 表示协议在哪里被处理。

## Rpc

定义Rpc。

<rpc name=”Trade” argument=”BTradeArgument” result=”BTradeResult”

handle=”server”/>

argument是一个Bean。result是一个Bean。handle 表示协议在哪里被处理。

## Project

定义项目，对应一个进程。一个solution一般拥有两个项目。一个server，一个client。

可能还有一个test。

<project name="GameServer" scriptdir="src" platform="java">

<project name="GameServer" platform="cs">

## Service

网络服务定义。协议在网络服务里面注册。网络服务管理连接以及提供网络事件和收到的协议的派发。

<service name="Server" handle="server”>

<module ref="Role"/>

</service>

module ref 引用模块。被一个Service引用的模块内定义的协议会被自动注册。

handle 引用模块内定义的协议的符合这个类别的，在这个服务里面注册和派发。

## Protocol.Handle & Rpc.Handle & Service.Handle

处理标签包含：server,client, serverscript, clientscript。采用server,client的叫法，仅仅因为这样比较符合网络程序功能通常的划分。支持多个标签，用”,”隔开。

## Type

Bean变量支持的类型以及在不同语言内的实际类型。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| type | Java | C# | Lua | TypeScript |
| bool | boolean | bool | boolean | boolean |
| byte | byte | byte | number(int64) | number |
| short | short | short | number(int64) | number |
| int | int | int | number(int64) | number |
| long | long | long | number(int64) | bigint |
| float | float | float | number(double) | number |
| double | double | double | number(double) | number |
| binary | Zeze.Net.Binary | Zeze.Net.Binary | string | Uint8Array |
| string | String | string | string | string |
| map | CollMap2<Bean>  CollMap1<Integer> | PMap2<Bean>  PMap1<int> | table | Map |
| list | CollList2<Bean>  CollList1<Integer> | PList2<Bean>  PList1<int> | table | Array |
| set | CollSet1<Integer> | PSet1<int> | table | Set |
| dynamic | DynamicBean | DynamicBean | table | DynamicBean |

# Use Arch

Arch详细说明和使用。

## Provider & Module

在Arch中，Provider是真正的逻辑实现服务，它躲在Linkd后面。用一套协议和Linkd交互。下面的Server是一个Provider的实现，用来实现主逻辑。Server后端有一个共享的数据库（如，mysql）。其他自定义的Provider实现Arch没有直接提供支持。自己根据需要使用Provider协议和Linkd交互即可。

1. 每个Provider是一个project，独立进程。
2. Module是逻辑划分单位，里面可以包含Bean，协议，数据（表）。Linkd转发的时候，同一个Module的协议会派发到同一个Provider实例。每个Module也可以看作是一个微服务。
3. 整个系统可以包含多个Provider。Provider包含多个Module实现。Module有一个唯一编号，在整个系统内唯一，即每个Module只能在一个Provider内实现。
4. 一个Provider可能存在多个实际运行进程。此时同一个Module会运行在多个进程中，但没有违反上一点。Module的唯一性是代码在Provider-Project中，实际运行可以是多进程。
5. Provider 的 Service.type 一般都是 "server"，但是它主动连接 linkd，并注册支持的Module。
6. 绑定亲缘性。当linkd给client选择provider-instance时，会把该Provider支持的Module全部都一起绑定到client.sesion中。

## Linkd

连接负载分配服务器。

### LinkdApp

管理Arch模块内部依赖类的引用。也完成一些初始化工作。

### LinkdProvider

Linkd面向内部Provider（GameServer是一个Provider）实现模块。包含负载选择，Provider协议的处理。Provider协议是定义内部服务怎么跟Linkd通讯的协议。

### LinkdProviderService

Linkd面向内部Provider的网络模块，管理跟内部Provider之间的网络连接和网络事件派发。

### LinkdService

Linkd面向客户端的网络模块，管理客户端连接。

## Linkd初始化

### 定义服务(solution.linkd.xml)

<project name="linkd" gendir="." scriptdir="src" platform="java">

<service name="LinkdService" handle="server"

base="Zeze.Arch.LinkdService">

<module ref="Linkd"/>

</service>

<service name="ProviderService" handle="client"

base="Zeze.Arch.LinkdProviderService">

</service>

</project>

Linkd服务需要使用Arch里面的两个服务，在base里面指定。如果应用需要对网络事件进行拦截处理，在生成的服务类LinkdService,ProviderService中重载相应的函数。Linkd服务作为一个项目，可以自由的增加自己的模块。参考 zeze/ZezeJava/Zege/solution.linkd.xml

### 配置（linkd.xml）

<?xml version="1.0" encoding="utf-8"?>

<zeze

GlobalCacheManagerPort="5555"

CheckpointPeriod="0"

ServerId="-1"

>

Linkd通常不需要数据库，但是验证可能要，先占个坑。

<DatabaseConf Name="" DatabaseType="Memory" DatabaseUrl=""/>

<ServiceConf Name="LinkdService"

InputBufferMaxProtocolSize="2097152"

SocketLogLevel="Trace">

<Acceptor Port="5100"/>

</ServiceConf>

<ServiceConf Name="ProviderService"

InputBufferMaxProtocolSize="2097152"

SocketLogLevel="Trace">

如果linkd运行在双网（内外网）机器上，这里可以配置Ip为内部网络的地址，不允许外部连接。

<Acceptor Ip="" Port="5101"/>

</ServiceConf>

<ServiceConf Name="Zeze.Services.ServiceManager.Agent">

<Connector HostNameOrAddress="127.0.0.1" Port="5001"/>

</ServiceConf>

</zeze>

### Linkd需要的Arch模块初始化

下面的初始化代码部分是生成的，应用需要加入Arch模块的初始化，当然也可以加入任意自己需要的初始化。Arch模块作为全局变量定义在App中。

public void Start(String conf) throws Throwable {

var config = Config.Load(conf);

// 生成的初始化

CreateZeze(config);

CreateService();

// Arch模块初始化

LinkdProvider = new Zeze.Arch.LinkdProvider();

LinkdApp = new Zeze.Arch.LinkdApp("Zege.Linkd", Zeze, LinkdProvider,

ProviderService, LinkdService, LoadConfig());

// 生成的初始化

CreateModules();

Zeze.Start(); // 启动数据库

StartModules(); // 启动模块，装载配置什么的。

// 设置Session生成器

AsyncSocket.setSessionIdGenFunc(PersistentAtomicLong.getOrAdd(

LinkdApp.GetName())::next);

StartService(); // 启动网络

// 注册服务。

LinkdApp.RegisterService(null);

}

## Server

Provider的主实现。一个App框架中只有一个主Server。在主Server中实现主要的应用逻辑。Arch框架不包括自定义的Provider实现。

### ProviderApp

管理Arch模块内部依赖类的引用。也完成一些初始化工作。

### ProviderImplement

处理Linkd转发的来自客户端的请求。客户端请求通过Provider协议包装。

### ProviderService

跟Linkd通讯的网络模块。

### ProviderDirect

处理Server之间互联协议。主要包括ModuleRedirect相关处理。

### ProviderDirectService

Server之间互联网络模块。

### ProviderModuleBinds

Server转发相关配置处理。

### Online

基于账号的在线管理模块。提供给任意在线用户发送消息的接口。

#### 本机数据

账号登录在某一个Server实例上时，可以保存一个仅当前Server可见的本机数据。本机数据每个登录保存一份。

* SetLocalBean 设置本机数据
* GetLocalBean 获取本机数据
* WalkLocal 遍历本机所有的数据，必须在事务外执行。

#### 在线事件

* LoginEvents登录发生时触发。
* ReloginEvents Relogin发生时触发。这个是可选的功能，如果应用不需要这个逻辑，不要在客户端发送Relogin协议即可。这个协议的功能和最小化数据同步相关。当客户端异常断线（没有主动发送Logout协议）时，服务器不会马上删除用户的在线状态，会保持一段时间。如果用户在这段时间内Relogin，就可以把这段时间内发生的数据变化同步给客户端，而且仅同步差异数据。这样就可以快速Relogin。这个功能需要同步差异数据，还需要配合ReliableNotify，ChangeListener一起使用。
* LogoutEvents登出时触发。登出事件可能丢失，在服务器关闭时补发一个Logout的处理非常困难（因为程序还可能异常关闭）。现在的逻辑是下一次Login发生时，发现上一个Login没有Logout，就补发一个额外的Logout事件。这个特性虽然完成了Login，Logout的匹配，但只能用于跟时间无关的逻辑中。如果逻辑需要Login，Logout匹配来完成时间统计，由于补发的Logout可能很迟，已经没法正确即使。这种需求需要注意，请自行处理。
* LocalRemoveEvents LocalData被删除时触发。

#### ReliableNotify（在线可靠消息）

在线可靠消息被保存在在线数据中，它发送给客户端并需要确认。没有确认的消息在Relogin时会被同步给客户端。

* AddReliableNotifyMark 从这个点开始，启用参数listenerName指定的消息，即调用了这个之后的，SendReliableNotifyXXX才真正生效。
* RemoveReliableNotifyMark 停止参数listenerName指定的可靠消息。
* SendReliableNotify 发送可靠消息。
* SendReliableNotifyWhileCommit 事务成功时发送可靠消息。
* SendReliableNotifyWhileRollback 事务失败时发送可靠消息。

#### SendToLogin（给某个登录发送消息）

给某个登录发送消息，只有这个登录能收到。

* SendEmbed 直接在当前事务中执行发送消息逻辑，回滚不会发送。
* Send 开启一个新的事务发送消息，可以在事务外使用。
* SendWhileCommit 事务提交时发送消息。
* SendWhileRollback 事务回滚时发送消息。

#### SentToAccount（给某个账号发送消息）

给某个账号发送消息，这个账号所有的登录都能收到。账号发送方法提供了回调函数。应用可以在回调中挨个处理每个登录，进行特殊的处理。没有提供回调函数时(参数是null)，默认实现给账号所有的登录发送消息。

* SendAccountsEmbed 直接在当前事务内执行发送消息，回滚不发送。
* SendAccount 开启一个新的事务给单个账号发送消息，可以在事务外使用。
* SendAccounts 开启一个新的事务给多个账号发送消息，可以在事务外使用。
* SendAccountWhileCommit 事务提交时发送。
* SendAccountsWhileCommit 事务提交时发送。
* SendAccountWhileRollback 事务回滚时发送消息。
* SendAccountsWhileRollback 事务回滚时发送消息。

#### Transmit

在分布式架构中，用户被分到了多台Server实例中。当用户A需要查询用户B的数据，发送给用户X（实际上通常X就是A）。虽然每个Server实例都能直接看到所有用户的数据，但是如果这个数据查询量大，并且改动非常频繁，那么此时从异机查询会导致Cache失效，命中率下降。所以提供了这个功能。

Transmit(string account, string clientId, string actionName, string target, Serializable parameter = null)

* **Sender**: Account,ClientId 请求发起者，数据结果发送给他。
* **actionName**：具体的查询操作，需要注册。
* **Target**：查询目标用户，
* **Parameter**：查询参数，可选。

Transmit会查找目标用户在哪台Server实例中，然后把请求转到他所在的server中执行。执行的结果直接发回给Sender，不会返回Sender所在服务器。这样，修改和查询都发生在Target用户所在的服务器，Cache命中率极高。而且整个操作代价不大，仅仅多一个Rpc转发。

* Transmit
* TransmitWhileCommit
* TransmitWhileRollback

#### Broadcast

***整个世界广播，慎用！***

### LoadConfig

负载配置。

### RedirectHash

这是一个注解，被标记的函数会被重定向到另一个Server进程执行。重定向使用第一个函数的参数hash进行选择。

@RedirectHash(ConcurrentLevelSource="getConcurrentLevel(arg1.getRankType())")

public RedirectFuture<Long> updateRank(int hash, BConcurrentKey keyHint, long roleId, long value, Binary valueEx)

ConcurrentLevelSource用来获得总的hash分组的数量。

Hash 选择依据，在上面这个例子，hash = roleId.hashCode();

#### 功能理解

虽然Zeze提供了分布式能力，但有时系统内有一些全局单点模块，不能提供足够的并发能力。举个例子，存在一种游戏内的即时排行榜，角色数值变化马上更新排行榜。由于这个排行榜只包含一个数据列表，所有的更新请求需要排队互斥进入列表。当同时在线角色数量很大，更新非常多，那么这个排行榜就会成为一个并发瓶颈。如果全局单点的数据可以按一定规则分开存储，并在需要的时候汇总。能这样做的数据仍然可以用很小的代价并提供足够的并发性能。比如上面的排行榜，分成128个分组数据，把角色hash分散到这些分组中。每个分组独立进行排名，每个分组都保存足够数量的排名。当需要全局排名时，把所有的分组整合起来就能得到最终的排名。这样排行榜的并发就增加了128倍。

#### 分组数量（ConcurrentLevelSource）

分组数量决定了最大的并发度。一般来说设置足够大，并留有一定余地即可。比如128。嗯，这个数字比较漂亮。分组数量一般来说不好随便改。比如对于排行榜来说，修改这个参数，对导致分组数据全部失效（需要作废掉重来）。

#### 负载分配

为了提高数据的Cache命中，访问同一个分组的请求需要转发到同一台服务器执行。分组数量是固定的。但服务器数量开始一般小于分组数量。RedirectHash会根据hash把负载分配到实际服务器中。每个服务器可能处理多个分组。当然分组数量决定了最大服务器数量。

### RedirectAll

Server之间直连协议的一种。给所有的分组数据发送广播请求，并处理结果（可能）。有点像MapReduce。这是RedirectHash的分组广播的注解，用来发送或者接收所有分组的数据。

@RedirectAll

protected RedirectAllFuture<RRankList> getRankAll(int hashCount, BConcurrentKey keyHint)

### RedirectToServer

这是一个注解，把标记的函数重定向到另一个Server进程执行。相当于一种便利的Rpc机制。

@RedirectToServer

public void redirectNotify(int serverId, String account) throws Throwable {

// some operate

}

ServerA里调用redirectNotify，实际会执行参数serverId所指定的服务器实例（有可能就是ServerA）。

## Server初始化

### 定义服务(solution.xml)

<project name="server" gendir="." scriptdir="src" platform="java" GenTables="">

<!--

这里引用的模块不该定义协议，定义了也不会被生成，一般为纯逻辑或者数据库模块。

<module ref="CommonModule"/>

-->

<!-- service 生成到 solution 名字空间下 -->

<service name="Server" handle="server" base="Zeze.Arch.ProviderService">

<module ref="User"/>

<module ref="Friend"/>

<module ref="Message"/>

</service>

<ModuleStartOrder>

</ModuleStartOrder>

<service name="ServerDirect" handle="server,client"

base="Zeze.Arch.ProviderDirectService">

</service>

</project>

Server需要使用Arch中的两个网络模块。在base中指定。需要拦截网路事件，可以在生成的类中重载相应函数。

### 配置(server.xml)

<?xml version="1.0" encoding="utf-8"?>

<!--

GlobalCacheManagerHostNameOrAddress: server 启用 cache-sync，必须指定。所有的 server 必须配置一样。

ServerId 每个 server 必须配置不一样，范围 [0, AutoKeyLocalStep)

AutoKeyLocalStep: 自增长步长。server 实例数量上限。

-->

<zeze

GlobalCacheManagerHostNameOrAddress="127.0.0.1" GlobalCacheManagerPort="5555"

CheckpointPeriod="60000"

ServerId="0"

>

先配一个内存数据库，调试。

<DatabaseConf Name="" DatabaseType="Memory" DatabaseUrl=""/>

<ServiceConf Name="Server"

InputBufferMaxProtocolSize="2097152"

SocketLogLevel="Trace">

</ServiceConf>

<ServiceConf Name="Zeze.Services.ServiceManager.Agent">

<Connector HostNameOrAddress="127.0.0.1" Port="5001"/>

</ServiceConf>

<ServiceConf Name="ServerDirect"

InputBufferMaxProtocolSize="2097152"

SocketLogLevel="Trace">

<Acceptor Ip="" Port="5102"/>

</ServiceConf>

</zeze>

### Server需要的Arch模块初始化

下面的初始化代码部分是生成的，应用需要加入Arch模块的初始化，当然也可以加入任意自己需要的初始化。Arch模块作为全局变量定义在App中。

public void Start(String conf) throws Throwable {

var config = Config.Load(conf);

CreateZeze(config);

CreateService();

// 初始化Arch模块。

Provider = new Zeze.Arch.ProviderWithOnline();

ProviderDirect = new Zeze.Arch.ProviderDirect();

ProviderApp = new Zeze.Arch.ProviderApp(Zeze, Provider, Server,

"Zege.Server.Module#", ProviderDirect, ServerDirect,

"Zege.Linkd", LoadConfig());

// 初始化可选的Arch模块

Provider.Online = GenModule.Instance.ReplaceModuleInstance(

this, new Online(this));

// 初始化可选的Zeze模块

LinkedMaps = new LinkedMap.Module(Zeze);

DepartmentTrees = new DepartmentTree.Module(Zeze, LinkedMaps);

// 生成的代码

CreateModules();

Zeze.Start(); // 启动数据库

StartModules(); // 启动模块，装载配置什么的。

// Online初始化

Provider.Online.Start();

// 其他初始化

PersistentAtomicLong socketSessionIdGen = PersistentAtomicLong.getOrAdd("Zege.Server." + Zeze.getConfig().getServerId());

AsyncSocket.setSessionIdGenFunc(socketSessionIdGen::next);

StartService(); // 启动网络

ProviderApp.StartLast(ProviderModuleBinds.Load(), Modules);

}

## 发现Linkd,Server流程

### Linkd.Startup

{

ServiceName = “Game.Linkd”; // 应用自定义。

ServiceIdentity = Ip + “:” + Port; // Linkd监听地址和端口，接受来自Provider的连接。

RegisterService(ServiceName, ServiceIdentity, Ip, Port, extra);

// Linkd一开始不会订阅Server的服务信息，在后面的发现流程中才会订阅。

}

### Server.Startup

{

// 每个模块作为一个服务。下面注册和订阅模块服务。

// 每个Server的CurrentModules是可以配置的。

Foreach (var Module in CurrentModules)

{

ServiceName = “Game.Server.Module#” + Module.Id; // 名字前缀应用自定义。

ServiceIdentity = Zeze.Config.ServerId; // Zeze服务器编号。

// ProvideIp, ProviderPort：Server监听地址和端口，接受来自其他Server的连接。

RegisterService(ServiceName, ServiceIdentity, ProvideIp, ProviderPort, extra);

SubscribeService(ServiceName, Module.SubscribeType);

}

// 订阅Linkd服务器列表

SubscribeService(“Game.Linkd”, Simple);

// 报告一次初始负载。

Load.Report(0, 0); // 0 online 0 new

}

### Server发现Linkd

1. Linkd.Startup向ServiceManager注册自己。
2. ServiceManager广播，Server会收到。
3. Server连接Linkd。
4. Server向Linkd发送Bind协议，马上绑定自己支持的静态模块。
5. Linkd收到Bind。完成模块绑定，并订阅(SubscribeService)该模块服务。

### 发现Server

1. Server.Startup向ServiceManager注册自己支持的所有模块服务。
2. ServiceManager广播，所有订阅者（Linkd和Server）都会收到。
3. Linkd收到Server列表变更不做任何处理，只是等待Server连接过来并处理Bind。
4. Server收到待定的列表时，开始连接新的服务器。
5. 连接建立后，主动方发送自己的地址和端口给被动方，设置本地状态为Ready。
6. 连接建立后，被动放收到主动方的地址和端口，设置本地状态为Ready。

## Session & UserState

1. AsyncSocket.UserState连接上下文，一般在连接创建的时候初始化，用来保存跟链接相关的自定义状态，用来实现Session功能。
2. Protocol.UserState 协议上下文，从某个连接收到的协议的上下文，默认设置为该连接的UserState。框架根据自己的需要可以设置新的Protocol.UserState。
3. Procedure.UserState存储过程上下文，为协议的处理而创建的存储过程，默认设置为该协议的UserState。框架可以根据自己的需要修改存储过程的UserState。

See ProviderImplement. ProcessDispatch

## Provider Protocol

### Bind

Provider发送给Linkd，绑定自己支持的Module。

1. 静态绑定

绑定时不指定客户端会话，绑定信息存储在Linkd的ProviderSession中。Linkd收到静态绑定内的模块的第一个协议请求时，根据这个模块的配置进行负载选择，选择一个Provider实例派发请求，同时所有静态绑定的模块都和客户端会话关联起来。以后所有静态绑定的模块的请求都会派发给这个Provider实例。

TODO静态绑定现在被主Server用来注册通用模块。目前静态绑定只有一个唯一的集合，所以不能跨越多个Provider。考虑支持多个集合的静态绑定。

1. 动态绑定

绑定模块到客户端会话。以后这个模块的所有协议都会被Linkd派发给这个Provider。这个模块也被称作动态模块。

### UnBind

Provider发送给Linkd，解除绑定。

### Subscribe

Provider发送给Linkd，通知Linkd订阅动态绑定的模块的服务信息。动态绑定的模块没有只能绑定到某个客户端会话，在绑定前，Linkd一无所知。某些Linkd的功能可能需要动态模块的服务信息。这时Provider可以通过Subsrcibe协议告知Linkd。这个能力是可选的，目前Arch实现了这点，但没有真正用到它，算是未来扩充Linkd功能的准备。

### Kick

Provider发送给Linkd，踢掉某个客户端会话。

### Send

Provider发送给Linkd，转发协议给指定的客户端。

### Broadcast

Provider发送给Linkd，向Linkd内所有客户端会话广播。

### SetUserState

Provider发送给Linkd，设置客户端会话的状态。这个状态以后在Linkd发送客户端会话相关的请求给Provider时，原样带上。这个功能一般用来实现LoginSession。

### Dispatch

Linkd发送给Provider，转发客户端请求。

### LinkBroken

Linkd发送给Provider，报告客户端连接断开。

### AnnounceLinkInfo

Linkd发送给Provider，通告Linkd的信息。这是一条保留协议，目前没有报告任何信息。

### AnnounceProviderInfo

Provider发送给Linkd，报告Provider的基本信息。

## 全系统维护启动顺序

1. 启动ServiceManage
2. 启动GlobalCacheManager
3. Linkd Server任意顺序启动。

## 全系统维护停止顺序

1. 停止ServiceManager 必须
2. Linkd.AcceptorClient.close() 阻止新用户进来
3. Server发送广播通知用户下线
4. Server等待一定时间后关闭
5. Linkd关闭
6. GlobalCacheManager关闭

# Bean

## Tree & Managed

Bean变量可以是一个Bean，Bean的变量可以是一个容器，容器内可以放入Bean。Bean通常放到Table中管理。这样从Table为根，Bean的变量最终组成了一颗树，Bean的实例不会被重复引用，也不会出现环。

Bean被加入Table就进入了Managed状态（或者被加入一个已经Managed状态的Bean的容器中），在Managed之前，修改Bean不会被记录日志。Managed状态一旦设置，就不会恢复，即使你从Table中或者容器中删除它。当你从Table或者容器中删除后要再次加入进去，需要Copy一次。Managed状态只能被设置一次。如果你想加入重复的对象，使用 Bean.Copy 方法复制一份。

## 兼容性

### Variable.Id

Bean可以自由的增删variable，新旧Bean在系列化时自动兼容。其中varible.id在一个Bean内不能重复。一般也不准回收理由删除的variable的id。删除variable，再恢复是允许的，此时恢复的variable定义必须完全一样，这种情况被看作反悔操作，而不是新定义的variable用了旧的id。反悔操作在原来的数据没有被覆盖的情况下，会重新读到旧的数据。一般反悔操作是开发过程中，但还没发布的情况下发生。

### 数值类型

所有的数值类型(byte,short,int,long,float,double)之间互相兼容(运行时透明自动转换)，转换规则跟所用编程语言的强转相同。所以注意高位截断等问题，通常应该从小范围类型改成大范围类型。

bool类型跟数值类型也互相兼容，bool转数值成0和1，数值转bool会用"!=0"来取结果。

### Binary & String

binary和string类型互相兼容，string的序列化是用UTF-8编码成binary，binary转string时如果遇到无法UTF-8解码时会抛异常。Lua的string因为无需编解码所以跟binary完全兼容。

### Dynamic & Bean

dynamic类型只是附带类型ID而不是根据配置选择具体类型名的bean，根据配置指定的方法用类型ID取得具体的bean类型再反序列化。如果改成bean类型，则忽略类型ID直接按指定的具体bean类型反序列化。如果从bean类型改成dynamic类型，则用默认0当作dynamic的类型ID。

### List & Set & Array

序列容器(list,set,array)互相兼容，容器内类型的兼容性同上。

### Map

关联容器(map)内key和value的类型兼容性按上面的规则处理。

### 一般兼容描述

bean类型的兼容性只看字段ID和字段类型，与bean的命名和类型ID无关。序列化数据里缺失的字段会当成默认值(0,false,空binary,空string,空容器,所有字段均为默认值的bean,内容是EmptyBean的DynamicBean)。如果序列化数据里有当前bean类型中不存在的字段ID，则直接忽略，再序列化时就会丢弃该字段。以上没提到兼容性和转换规则的转换均不支持，遇到则用默认值取代旧值。

## BeautifulVariableId

关于bean增减variable的建议：按variable.id的顺序从1开始自增地分配和扩展；删除variable不要直接删除，可以修改variable.name或注释来表示“临时不再使用”的含义，方便保留数据库中已有的数据不丢失，以备之后再恢复使用，也防止增加variable时重用该variable.id引发取出旧数据的混乱。如果有彻底全服删库的机会，可以删除不会再用的variable，此时也可以顺便重新整理所有的variable.id。

但是开发过程中由于实现方案的修改，已经确定不需要旧的variable时（留着碍眼），可以删除variable。再次提醒，除非反悔，不要重新利用旧的variable.id。由于开发期不稳定，被删除的variable比较多，造成id不连续。对于有洁癖的程序员，就可以考虑利用Gen.exe -BeautifulVariableId 功能重置id，使得他们从一开始连续编号。这个操作是个不兼容的修改，只能用于开发期删除所有数据库数据的时候一并处理。一旦程序发布，就不能再使用BeautifulVariableId功能了。使用BeautifulVariableId之后需要重新生成相关代码。

## Version

Table.Value可以定义一个版本号。当修改发生时自动递增。Table.Value总是一个Bean，这个定义方式就放到这里了。声明例子：

<bean name=”BVersionSample” version=”VersionVarName”>

<variable id=”1” name=” VersionVarName” type=”long”/>

…

</bean>

1. VersionVarName 可以自己命名，然后在Bean的属性version中声明哪一个变量用来存储版本号。
2. Type 必须是long。
3. 使用Bean.GetVersion()方法得到当前的版本号。
4. 如果Bean没有作为Table.Value，那么GetVersion总是返回0，没有意义。
5. 如果Table.Value记录被删除，然后再次加入，那么版本号重新从0开始。

# Serialize

Zeze有一个很小的自己的系列化实现。麻雀虽小五脏俱全。

主要如下两个类：

1. Zeze.Serialize.Serializable 接口
2. Zeze.Serialize.ByteBuffer 实现

Zeze.Transaction.Bean是在ByteBuffer基础上构建的一套对象系列化实现。

## 类型枚举(4-bit)

0: 有符号整数(byte,short,int,long,bool)

1: 单精度浮点数(float)

2: 双精度浮点数(double)

3: 二进制数据/字符串(binary,string)

4: 序列容器(list,set)

5: 关联容器(map)

6: bean

7: 动态bean(dynamic)

8~15: 可自行扩展定义非标准类型

## 整数压缩

比如：用户定义了long类型，当值小于128时，仅用一个字节保存。

\* 有符号整数(支持64位补码有符号整数的所有值)

正整数:

1字节(< 0x 40): 00xx xxxx (取低有效位,按大端排列)

2字节(< 0x 2000): 010x xxxx +1B

3字节(< 0x 10 0000): 0110 xxxx +2B

4字节(< 0x 800 0000): 0111 0xxx +3B

5字节(< 0x 4 0000 0000): 0111 10xx +4B

6字节(< 0x 200 0000 0000): 0111 110x +5B

7字节(< 0x 1 0000 0000 0000): 0111 1110 +6B

8字节(< 0x80 0000 0000 0000): 0111 1111 0xxx xxxx +6B

9字节( unlimited): 0111 1111 1xxx xxxx +7B

负整数:

1字节(>=-0x 40): 11xx xxxx

2字节(>=-0x 2000): 101x xxxx +1B

3字节(>=-0x 10 0000): 1001 xxxx +2B

4字节(>=-0x 800 0000): 1000 1xxx +3B

5字节(>=-0x 4 0000 0000): 1000 01xx +4B

6字节(>=-0x 200 0000 0000): 1000 001x +5B

7字节(>=-0x 1 0000 0000 0000): 1000 0001 +6B

8字节(>=-0x80 0000 0000 0000): 1000 0000 1xxx xxxx +6B

9字节( unlimited): 1000 0000 0xxx xxxx +7B

\* 无符号整数(支持32位无符号整数的所有值)

仅用于序列化长度,数量,增量等特定情况

1字节(<0x 80): 0xxx xxxx (取低有效位,按大端排列)

2字节(<0x 4000): 10xx xxxx +1B

3字节(<0x 20 0000): 110x xxxx +2B

4字节(<0x1000 0000): 1110 xxxx +3B

5字节( unlimited): 1111 0000 +4B

## 单精度浮点数(float)

按IEEE754标准序列化成小端排列的固定4个字节

## 双精度浮点数(double)

按IEEE754标准序列化成小端排列的固定8个字节

## 二进制数据(binary)

+ 无符号整数: 数据的字节长度

+ 数据的原始内容

## 字符串(string)

+ 按二进制数据序列化字符串的UTF-8编码数据

## 序列容器(list,set)

+ 单字节: (高位)nnnn tttt(低位)

t: 容器元素的类型枚举

n=0~14: 容器元素的数量

n=15: 附加一个无符号整数(x),用15+x表示容器元素的数量

+ 按以上指定类型及枚举顺序连续序列化容器的所有元素

## 关联容器(map)

+ 单字节: (高位)kkkk vvvv(低位)

k: 容器键(key)的类型枚举

v: 容器值(value)的类型枚举

+ 无符号整数: 容器键值对的数量

+ 按以上指定类型及枚举顺序连续序列化容器的所有键值, 即"键值键值..."的顺序

## 字段标签(tag)

+ 单字节: (高位)iiii tttt(低位)

i=0: 特殊含义

t=0: 结束标签

t=1: 结束当前层的标签,后续是上一层(父类)的序列化

t=2~15: 未定义,保留扩展

i=1~14: 距上个字段ID的增量(初始为首个字段ID), t表示字段的类型枚举

i=15: 附加一个无符号整数(x),用15+x表示ID增量, t表示字段的类型枚举

## bean

+ 按"字段标签,字段值,字段标签,字段值,...,结束标签(单字节0)"的顺序序列化

序列化字段的顺序要求按字段ID从小到大排列, 字段ID的支持范围: [1,0x7fffffff]

字段值如果等于默认值,可省略该字段标签及其值的序列化

反序列化时要求先重置bean的所有字段为默认值再反序列化

有继承关系的bean要求先序列化子类字段,然后插入结束当前层的标签,再序列化上一级父类

默认值的定义: 数值=0; 二进制数据长度=0; 字符串长度=0; 容器元素数量=0;

bean的所有字段均为默认值; 动态bean为未定义值(空值)

## 动态bean(dynamic)

+ 有符号整数: 动态bean的ID

+ bean的序列化

\* 协议帧(protocol)

+ 小端排列的固定4字节无符号整数: 模块ID(moduleId)

+ 小端排列的固定4字节无符号整数: 协议ID(protocolId)

+ 小端排列的固定4字节无符号整数: 下面bean的序列化字节长度

+ bean的序列化

## 关于序列化的兼容性

有符号整数,单精度浮点数,双精度浮点数之间可以自动转换,但要注意高位截断和降低精度问题

binary和string之间可以互相转换,但要注意binary转到string时有无法正确解码UTF-8而抛出异常的可能

list和set之间可以互相转换,但要注意list转成set后再序列化可能会改变顺序

bean可以转成动态bean,默认typeId=0; 动态bean可以转换成bean,但要注意不同bean类型之间的字段兼容性

以上没提到的类型转换说明不兼容,反序列化会自动忽略不兼容的字段

# Net

Zeze有一个很小的自己的网络实现。麻雀虽小五脏俱全。

Net是一个异步模式的网络实现。发送数据不会阻塞，立即返回。接收数据由底层解码成协议并派发到线程池执行。

## AsyncSocket

对系统网络层进行必要的包装，它向应用提供发送数据的接口。

## Service

创建AsyncSocket。

管理AsyncSocket；

管理网络配置；

处理它的网络事件；

管理自己能处理的协议；

## 继承Service

Zeze应用通过继承Service的方式使用Net模块。

1. ***Service所有的处理函数都在网络线程中直接处理。所以，重载实现不能阻塞。***
2. ***Service的重载函数通常需要调用基类的方法，除非特殊情况。***

## Protocol

Zeze协议带有一个Argument。Argument是一个Bean。

编码规范：

Header=(ModuleId,ProtocolId)

ParameterSize

ParameterBinary=(ResultCode,EncodedArgument)

Decode：解码实现方法。

## 接收数据处理流程（Dispatch）

1. AsyncSocket 从系统接收数据
2. 进行解密解压（可选）
3. Call Service.OnSocketProcessInputBuffer（可重载实现自己全新的协议规范）
4. Call Protocol.Decode
5. Call Protocol.Dispatch（手动编写的协议子类可重载，自动生成的协议代码总是会被覆盖，没法写重载实现代码，这个重载就没有意义）
6. Call Service.DispatchProtocol（***协议派发主要重载接口，有默认实现***）

## Rpc

Rpc有Argument,Result两个参数。

Rpc is Protocol。

编码规范：

Header=(ModuleId,ProtocolId)

ParameterSize

ParameterBinary=(IsRequest,SessionId,ResultCode,ArgumentOrResult)

Rpc提供同步等待、异步回调两种方式处理结果。

## Connector

连接器。

## Acceptor

Acceptor

## ServiceConf

<ServiceConf Name="Zeze.Services.ServiceManager.Agent">

<Connector HostNameOrAddress="127.0.0.1" Port="5001"/>

</ServiceConf>

Name 是Solution.xml里面配置的名字。

Connector 连接器配置。可以包含多个。

Acceptor 可以和Connector一起存在，这个例子没有啦。

对一个Service来说，不管是来自Connector的连接还是来自Acceptor的连接，都享受一样的服务。

# Raft

## 什么是Raft

请阅读 zeze/Zeze/Raft/下的Raft.mhtml，raft.pdf，OngaroPhD.pdf。

这里不再重复。

## Zeze.Raft.StateMachine

应用数据定义在这里。

## Zeze.Raft.Log

定义应用数据（StateMachine）修改操作。

每次操作对应一个Log子类。

## Zeze.Raft.Raft.AppendLog

把修改操作日志（Log）添加到Raft日志队列中。

当AppendLog方法返回时，表示操作已经被成功处理。

## Zeze.Raft.Agent

应用的客户端管理类。

自动切换Raft-Leader变更，自动重发请求。

## Zeze.Raft.RaftRpc

自定义应用访问协议基类。

当协议用zeze生成的时候，在协议定义里面指名 base="Zeze.Raft.RaftRpc"。

建议每个RaftRpc对应一条Log。

ResultCode==Zeze.Transaction.Procedure.RaftApplied：Raft发现请求是重发的，但是已经成功处理过。

ResultCode==Zeze.Transaction.Procedure.RaftExpired：请求过期了，无法判断是否被成功处理。

## 例子

// 注意，以下例子没有处理多线程问题。

// 应用数据

class MyStateMachine : Zeze.Raft.StateMachine

{

public long Count;

// 操作日志

class AddCount : Zeze.Raft.Log

{

public AddCount(IRaftRpc req) : base(req)

{

}

public override void Apply(RaftLog holder, StateMachine stateMachine)

{

(stateMachine as MyStateMachine).Count += 1;

}

}

// 应用操作接口。这里传入的是下面定义的AddCount网络协议。

public void AddCount(IRaftRpc request)

{

Raft.AppendLog(new AddCount(request));

}

// 需要实现的接口

// 参考Zeze.Raft.Test.TestStateMachine

public override void LoadSnapshot(string path)

{

}

public override bool Snapshot(string path, out long LastIncludedIndex, out long LastIncludedTerm)

{

}

public MyAppStateMachine()

{

// 注册Log工厂。

AddFactory(new AddCount(null).TypeId, () => new AddCount(null));

}

}

// 增加Count计数的协议

public sealed class AddCount : RaftRpc<EmptyBean, EmptyBean>

{

public readonly static int ProtocolId\_ = Bean.Hash32(typeof(AddCount).FullName);

public override int ModuleId => 0;

public override int ProtocolId => ProtocolId\_;

}

// 服务器创建Raft实例

var configFileName = "raft.xml";

var config = Zeze.Raft.RaftConfig.Load(configFileName);

var nodeName = config.Name; // 所有的raft-node共享一个配置文件时，需要通过参数指定启动的node名字。

var stateMachine = new MyAppStateMachine();

var raft = new Raft(stateMachine, nodeName, config);

// 服务器协议处理

long ProcessAddCountRequest(Protocol p)

{

var r = p as AddCount;

stateMachine.AddCount(r);

}

// 客户端创建Agent

var agent = new Agent("MyRaftApp.Agent", config);

agent.Client.AddFactoryHandle(new AddCount().TypeId, new Net.Service.ProtocolFactoryHandle()

{

Factory = () => new AddCount(),

});

// 客户端发送网络请求

var req = new AddCount();

agent.SendForWait(req).Task.Wait();

if (req.Result.ResultCode == )

...

# Platform

Zeze支持多个开发平台。platform是project属性，需要明确指定。

## java

服务器开发平台。

## cs

服务器或者客户端。

## ts

纯TypeScript环境。一般是ts客户端，不包括完整环境。

## cs+ts

客户端使用Unity+TypeScript(依赖puerts)

1. 把 zeze/Zeze 发布到你的项目，直接拷贝代码或者需要自己编译发布二进制。
2. 把 zeze/TypeScript/ts/ 下的 zeze.ts 拷贝到你的 typescript 源码目录。
3. 依赖：npm install https://github.com/inexorabletash/text-encoding.git
4. 把 zeze/Zeze/Services/ToTypeScriptService.cs 文件中 #if USE\_PUERTS 宏内的代码拷贝到你的c#源码目录下的ToTypeScriptService.cs 文件中。当然这里可以另起一个文件名。
5. 把 typeof(ToTypeScriptService) 加到 puerts 的 Bindings 列表中。
6. 然后使用 puerts 的 unity 插件菜单生成代码。
7. 定义 solutions.xml 时，ts客户端要处理的协议的 handle 设置为 clientscript.
8. 使用 gen 生成协议和框架代码。
9. 例子可以看看 https://gitee.com/e2wugui/zeze-unity.git
10. 不知道怎么发布依赖，现在测试运行是把encoding.js encoding-indexes.js 拷贝到output下。其中 encoding.js 改名为 text-encoding.js。

## cxx+ts

客户端使用Unreal(cxx)+TypeScript(依赖puerts)

1. 把zeze\cxx下的所有代码拷贝到你的源码目录并且加到项目中。除了Lua相关的几个文件。
2. 把 zeze/TypeScript/ts/ 下的 zeze.ts 拷贝到你的 typescript 源码目录。
3. 依赖 npm install https://github.com/inexorabletash/text-encoding.git
4. 安装puerts，并且生成ue.d.ts。
5. 定义 solutions.xml 时，ts客户端要处理的协议的 handle 设置为 clientscript.
6. 使用 gen 生成协议和框架代码。
7. zeze\cxx\ToTypeScriptService.h 里面的宏 ZEZEUNREAL\_API 改成你的项目的宏名字。
8. 例子 https://gitee.com/e2wugui/ZezeUnreal.git
9. 不知道怎么把依赖库(text-encoding)发布到unreal中给puerts用，可以考虑把encoding.js encoding-indexes.js
10. 拷贝到Content\JavaScript\下面，其中 encoding.js 改名为 text-encoding.js。

## lua

纯lua环境，不包括完整框架。

## cs+lua

客户端使用Unity(csharp)+lua

1. 需要选择你的Lua-Bind的类库，实现一个ILua实现

（参考 Zeze.Service.ToLuaService.cs）。

1. 定义 solutions.xml 时，客户端要处理的协议的 handle 设置为 clientscript.
2. 使用例子：zeze\UnitTestClient\Program.cs。

## luaclient

纯lua环境。其中bean在lua中使用变量名字作为table的key。

## cs+luaclient

cs+lua特别版。参考cs+lua。

## cxx+lua

客户端使用Unreal(cxx)+lua

1. 依赖lualib, 需要设置includepath
2. 直接把cxx下的所有代码加到项目中。除了ToTypeScript相关的。
3. 定义 solutions.xml 时，客户端要处理的协议的 handle 设置为 clientscript.
4. 使用例子：zeze\UnitTestClientCxx\main.cpp

## conf+cs

cs系列化的可独立发布的版本，不依赖Zeze库。目前用于unity配置。这个版本的bean生成代码很简洁。

## conf+cs+net

cs系列化的可独立发布的版本，不依赖Zeze库，带了网络和日志增量同步。这个版本的bean生成代码很简洁。导出源码的批处理程序参考zeze/Zege/confcsnet.bat。一般项目准备步骤如下：

1. 定义自己solution.xml，并生成。
2. 把confcsnet.bat拷贝到将要独立发布的项目目录下并执行。
3. 在将要独立发布的项目配置中加上宏“USE\_CONFCS”。

# Component

Component是Zeze内建模块；

## DelayRemove

remove(TableX<K, ?> table, K key)

可以直接使用这个静态方法，或者通过Table.delayRemove使用。通过这里的remove删除的记录只会被加入延迟队列，然后在过一段时间才会被真的删除。这个功能最初的原因是为了解决基于记录的队列（Zeze.Collections.LinkedMap）的记录锁外并发遍历和删除的问题。LinkedMap用记录存储Node，并在Node中记录前后Node的记录的Key，每个Node存储一定量的Item。这个LinkedMap可以存储很大的Item数量。当遍历LinkedMap时，不能在一个事务内访问所有的记录（即Node），必须挨个处理记录，这样当下一个记录刚好被删除，由于当前Node还指向被删除的记录，那么如果马上删除记录，获取下一个记录时，会得到null，就不能继续遍历了。有了DelayRemove，当LinkedMap要删除记录时，先更新前后记录的索引，更新后执行的遍历只遍历新的记录，并发的遍历也能继续进行。一定时间以后，这个记录才会被真正删除。这个可以看作是一个简单垃圾回收机制。

这个模块不需要初始化。

## AutoKey

AutoKey用来分配系统内唯一的Id（long类型）。

仅保证唯一，不保证顺序。

这个模块由Zeze初始化。可通过Zeze.Application.AutoKeys.GetOrAdd(autoKeyName)得到AutoKey实例，然后访问实例的方法NextId得到下一个Id。保存GetOrAdd的返回值，以后继续使用，可以稍微提高点效率。

## RedoQueue

Zeze系统内提供了事务。但实际服务器架构中可能存在非Zeze的系统，而且其中的某些操作需要跨系统的事务支持。目前Zeze没有向外部系统提供事务支持。在符合一定条件下，可以用一个简单机制（而不是跨系统的事务）来达到目的。举个游戏的例子，一般游戏架构中有个独立的账号充值系统，完成充值的时候需要在游戏（ZezeApplication）内给玩家发放物品，这样的操作是可以重做的，只要游戏没有返回成功，充值系统可以一直重试。这就是RedoQueue完成的功能。RedoQueue提供了一个可自定义任务内容的框架，完成任务同步和重试。

RedoQueue分成两个部分。

1. RedoQueue(Client) 部分给非ZezeApplication用，它把加入队列的任务直接本地存储到RocksDb中，同时把任务发送给ZezeApplication进行处理。
2. RedoQueueServer 部分在ZezeApplication中执行。具体任务执行的操作需要应用自己注册。RedoQueueServer记录了队列的已完成任务编号，如果出现了回档（已完成任务编号变成以前的），队列会从当前没有完成的任务编号开始，继续处理整个队列。这就是叫做Redo的原因。

RedoQueue主要接口

void add(int taskType, Zeze.Serialize.Serializable taskParam)

taskType, taskParam完全由应用自己定义。

RedoQueueServer主要接口

void register(String queue, int type, Predicate<Binary> task)

queue 队列名字，一个RedoQueueServer支持多个Client。

RedoQueueServer初始化

MyApp.RedoQueueServer = new RedoQueueServer(zeze);

MyApp.RedoQueueServer.Start();

## Timer

### Auto Named Timer

String Schedule(delay); 一定延迟后执行一次。返回TimerId。

String Schedule(delay, period); 一定延迟后开始按间隔执行。

Striing Schedule(cron); crontab风格定时器配置。

1. 自动命名的定时器。每次调用调度函数都会注册新的定时器，内部自动命名。自动命名的定时器的名字以‘@’开头。
2. 持久化。重启以后Timer会继续调度。
3. 分布式。一开始在注册所在的Server上执行，当Server非法宕机会被调度到其他Server上。
4. 真正的调度用ScheduledThreadPool.schedule实现。

### Named Timer

boolean ScheduleNamed(string name, ...);

全局Timer，每个名字只有一份实例。如果注册时改名字已经存在，名字存在会返回false。一般来说这个timer跑在注册它时所在的server实例上。但有可能被调度到其他server实例上。

### Online Timer

String ScheduleOnline(userid, …);

这些Timer和用户绑定，支持账号或者RoleId。仅在满足相关在线状态时才生效，具有一定的自动生命期管理。

1. 仅在在线时允许注册，下线全部自动失效。
2. 非持久化！在线timer是把保存在内存里的，直接使用语言自带的Timer调度器实现。生命期跟随ModuleOnline.LocalData走。有这样的特性：用户登录在Server A上；没有正常下线(Logout)；又登录到Server B上；此时Server B会向Server A发一个Kick Local；然后Server A上的online-timer也会结束。这个流程不是原子的，仅是最终保证；也就是说某个特别的时刻，AB上同时都有该timer。
3. 数量不限！由登录在一个server进程上的最高在线用户自动形成限制。
4. 仅存在于登录所在的Server。
5. 生命期和ModuleOnline.LocalData一致。

### Offline Timer

String ScheduleOffline(userid, …);

这些Timer和用户绑定，支持账号或者RoleId。仅在满足相关在线状态时才生效，具有一定的自动生命期管理。

1. 仅在下线时允许注册，上线全部失效。一般在登出事件中，注册所有的Offline Timer。下线期间允许继续注册，但估计不常见。
2. 持久化！
3. 数量有限制。每个用户用一个Offline Bean存储它所有的定时器。
4. 持续限制！Offline Timer不能一直持续，需要次数或者时间限制。
5. 只允许在一台Server“下线”，即Offline Timer都在一台上注册和调度。当在下线在ServerA，然后又登录到ServerB时，ServerA的Timer会被去取消。
6. 生命期和ModuleOnline.LocalData相反。Offline Bean在ModuleOnline的登录事件中删除，内嵌到登录事务中，这样可以保持数据一致性。但取消ThreadPool的任务通过@Redirect通知。为了处理Redirect可能丢失的问题，需要在Offline Timer中记录Login.Version，并且在触发定期回调时检查定时器版本号是否和当前登录版本号一致。

# Collections

Collections是Zeze的内建模块。

## LinkedMap

一个容量巨大的Map。一个有顺序的Map，注意，不是排序的Map。

所有数据操作支持事务。

遍历操作是事务外的。

初始化

MyApp.LinkedMapModule = new LinkedMap.Module(zeze);

使用例子

Var friends = LinkedMapModule.open(“Friends”, BFreind.clsss);

|  |  |
| --- | --- |
| getRoot | 得到LinkedMap的根节点 |
| getNode | 查询数据节点 |
| getFristNode | 查询第一个数据节点 |
| isEmpty | 是否空 |
| size | 项数量 |
| moveAhead | 把指定项移到列表开头 |
| moveTail | 把指定项移到列表尾巴 |
| getOrAdd | 根据key查询项，没有就创建一个 |
| put | 插入或者替换一个key-value项 |
| get | 查询项 |
| remove | 移除项 |
| removeNode | 移除节点 |
| clear | 清楚整个容器 |
| walk | 遍历容器 |

## Queue

一个容量巨大的单向链表队列。

所有数据操作支持事务。

遍历操作是事务外的。

不需要初始化，通过Zeze.Application.Queues创建和大概队列。

|  |  |
| --- | --- |
| isEmpty | 队列是否为空 |
| pollNode | 删除并返回整个第一个节点，不存在返回null |
| peekNode | 返回第一个节点，不存在返回null |
| clear | 清除队列 |
| poll | 从队头提取项，没有则返回null |
| peek | 查询队头，队列为空返回null |
| size | 项的数量 |
| add | 在队尾添加项 |
| push | Stack |
| pop | Stack |
| walk | 事务外遍历整个队列 |

## DepartmentTree

这是一棵树。

初始化

MyApp.DepartmentTree = new DepartmentTree.Module(zeze, LinkedMapModule);

|  |  |
| --- | --- |
| getRoot | 返回管理树的根节点 |
| getDepartmentTreeNode | 返回部门节点 |
| getDepartmentMembers | 返回部门成员的LinkedMap |
| getGroupMembers | 返回树的所有成员的LinkedMap |
| selectRoot | 事务外，返回树的根管理节点 |
| selectDepartmentTreeNode | 事务外，返回部门节点 |
| create | 创建树 |
| changeRoot | 改变树的总管理员 |
| getOrAddRootManager | 树的根节点管理员 |
| getOrAddManager | 部门几点的管理员 |
| createDepartment | 创建部门 |
| deleteDepartment | 删除部门 |
| isRecursiveChild | 是否某个部门的子（孙）部门 |
| moveDepartment | 移动部门 |

# Services

Zeze内建服务。

都是独立进程。

## ServiceManager

管理服务器信息的注册和订阅。当新增服务器或者服务器关闭时，会给订阅者发送新的服务器列表。

|  |  |
| --- | --- |
| 名词 | 说明 |
| ServiceName | 服务名字 |
| ServiceInfo | 服务信息，包含ServiceName, ServiceIdentity,Ip,Port,ExtraInfo |
| ServiceIdentity | 服务编号，用来区分不同的服务器，具体含义由应用自己决定。 |
| Ip，Port | 服务器地址和端口，当服务作为Acceptor时一般需要提供。可选。 |
| ExtraInfo | 应用自定义数据，使用Binary存储。可选。 |
|  |  |

数据结构描述

Map<ServiceName, List<ServiceInfo>>

List 服务器列表按 ServiceInfo.ServiceIdentity排序。

### 订阅模式

#### Simple

某个ServiceName下的服务器信息列表发生改变的时候，给所有订阅者发送通知。

#### ReadyCommit

订阅者收到的服务器列表的时间有先后，还需要进行准备，如建立连接并登录等。这个过程不会很快，造成每个订阅者真正可使用的服务器列表不会一致。当订阅者使用hash方式选择服务器时，这种不一致会导致选中的服务器不一样。为了降低这种不一致，引入这种订阅模式。步骤如下：

1. 某个ServiceName下的服务器信息列表发生改变的时候，给所有订阅者发送通知
2. 开始收集订阅者的Ready回复。
3. 所有订阅者都Ready了以后，再通知一次准备好的服务器列表。

### ServiceManager.Agent

ServiceManager客户端，主要接口如下：

|  |  |
| --- | --- |
| 接口 | 说明 |
| RegisterService | 注册服务信息 |
| UnRegisterService | 撤销注册服务信息 |
| UpdateService | 更新服务信息 |
| SubscribeService | 订阅 |
| UnSubscribeService | 撤销订阅 |
| SetLoad | 设置负载信息，信息键值为”Ip:Port”。订阅服务信息时，如果信息中包含Ip,port，那么就自动订阅该服务器的负载。任何服务器都可以发送SetLoad。负载会被广播给关注这个ip:port的订阅者。目前没有撤销订阅的接口，只有当ServiceManager广播发现网络断开，才自动撤销负载订阅。 |

## GlobalCachaManager

一致性缓存锁管理服务器。

## GlobalCachaManagerWithRaft

Raft版本的一致性缓存锁管理服务器。

# Game

Game是Zeze内建模块。

## Online

基于角色的在线管理模块。提供给任意在线角色发送消息的接口。

主要功能跟Arch/Online差不多，这里不详细说明了。

## Bag

基于格子的包裹模块。支持堆叠，拆分，移动，合并等操作。

## Rank

在概念上，排行榜是对定义的数据进行排序。这种实现方式在大多数情况下都无法提供足够的性能。有一种实现是每天用备份库排一次序。这种的缺点狠明显，更新不及时，而且在数据量大时，仍然消耗大量计算资源。一般来说，排行榜只需要显示排在开头的少量数据，比如前100名。所以有一种实现是在排行榜相关数据变更时，马上更新排行榜（快速判断是否进榜）。对于只增长的数据，这种实现方式是完全正确的。对于可能变小的数据，有个缺陷，当排到当前排行榜数据的末尾时，可能是不正确的。因为原来没进榜的数据可能比此时末尾的大。这里一般用一个技巧，就是排行榜的个数在计算时比需求大些，比如1000个。这样当它处于第1000个时，从计算数据中删除它。只要计算数据中的数量还剩的比100多，仍然是正确的。如果比100还少，那么就显示少一些的数据，也不会出现玩家发现自己该进榜而不在里面的问题。这种实现方式，排行榜的写操作很很多，而且概念上还是全局共享的，需要额外的解决方案。

## ProviderDirectWithTransmit

Server之间直连处理模块的一个更具体实现，包含Transmit功能。

依赖ProviderImplementWithOnline。

## ProviderImplementWithOnline

Server处理来自Linkd转发的客户端请求模块的更具体实现，包含Online模块的实例，多了一个处理（ProcessLinkBroken），处理来自Linkd的客户端连接断开的事件。

# Web（Netty）

可嵌入Server的Web模块，提供开发简单的Web服务。这样系统就有了让浏览器成为终端的能力。对于一个有点规模的系统，这个能力是比较重要的。至少系统的在线管理功能是很适合使用浏览器的。这个模块包装了Netty提供的Http解析和网络服务，主要的实现都在名字空间Zeze.Netty下面。下面介绍主要的类和功能。

## Netty

Netty的启动关闭以及服务管理。主要方法有addServer, start, stop。

## HttpServer

Web网络服务管理类，实现Netty接口ChannelInitializer<SocketChannel>，初始化和处理Web请求的派发。请求派发给HttpExchange处理。主要方法addHandler。

## HttpExchane

HttpExchange是核心包装类，是Zeze-Web应用主要使用的类。它实现Netty Http请求聚合，简化上层的使用。另外提供一些发送Web-Response的便利辅助函数。

## HttpHandler

Web请求处理接口。包含如下的定义。

public int MaxContentLength = 8192; // -1 表示不限制，按流处理。

public TransactionLevel Level = TransactionLevel.Serializable; // 事务级别

public DispatchMode Mode = DispatchMode.Normal; // 线程派发模式

// 1. Request-Response处理模式

// 普通请求处理函数，不是流处理方式时，如果需要内部会自动把流合并到一个请求里面。

public HttpFullRequestHandle FullRequestHandle;

// 2. 异步流处理模式。

// 上行流处理函数。

public HttpBeginStreamHandle BeginStreamHandle;

public HttpStreamContentHandle StreamContentHandle;

public HttpEndStreamHandle EndStreamHandle;

说明：

* 不同处理模式实现相应的Handle。
* 这个Handler可以手动创建并注册到HttpServer中。
* 这个Handler也可以在solutions.xml中定义，把相应Handle映射到模块的实现类中。
* 注册Handler需要指定一个key，即web-uri-path。
* Web-uri-path注册只支持一一对应，不支持往path-parent方向搜索。
* 入门例子Zeze.Netty.Netty.java::main

# Limit

## Database

* Zeze.Transaction.Database.eMaxKeyLength

Table.Key的最大长度。默认为3072，这个数字来源自Mysql 8的varbinary为primary key时的最大长度。现在这个配置用于Sqlserver，Mysql以及兼容Mysql的数据库。修改这个参数需要重新编译Zeze。已经创建的表的Key类型需要自己手动修改(Alter)。

# 杂项

## 多数据库支持

Zeze支持配置多个后端数据库，向应用提供逻辑上的一个数据库，支持跨数据库事务。在zeze.xml中为每个数据库提供一个DatabaseConf 配置。多个数据库需要用不同 Name 区分。然后在 TableConf 中使用属性 DatabaseName 把表格分配到某个数据库中。一个表只能属于一个数据库。

## 从老的数据库中装载数据

当使用某些嵌入式数据库（比如bdb）时，如果某个数据库文件很大，但是活跃数据可能又不多，每次备份它比较费时。可以考虑把表格移到新的数据库，然后系统在新库中找不到记录时，自动从老库中装载数据。这样，老库是只读的，不用每次备份。TableConf 中使用属性 DatabaseOldName 指明老的数据库，同时属性 DatabaseOldMode 设为 1。可以很好支持增量备份的数据库不需要使用这个特性。

## 同一个进程内多个Zeze.Application之间的事务

XXX 已经被废弃，不再支持，需要的喊！写在这里作为备注。

一般来说，事务仅仅访问一个 Zeze.Application 的数据库表格。如果需要在多个 Zeze.Application 之间支持事务。应用直接访问不同 App.Module。里面的表格即可完成事务支持。不过由于事务提交(Checkpoint)默认是在一个 Zeze.Application中执行的，为了让事务提交也原子化。需要在App.Start前设置统一Checkpoint。

设置代码例子：

Zeze.Checkpoint checkpoint = new Zeze.Checkpoint();

// 把多个App的数据库加入到Checkpoint中。

checkpoint.Add(demo1.App.Zeze.Databases.Values);

checkpoint.Add(demo2.App.Zeze.Databases.Values);

// 设置App的Checkpoint。

demo1.App.Zeze.Checkpoint = checkpoint;

demo2.App.Zeze.Checkpoint = checkpoint;

// 启动App。必须在启动前设置。

demo1.App.Start();

demo2.App.Start();

## 统计

Zeze统计了几乎所有各种可能的情况。c#可以通过宏完全关闭统计。这个主要为了以后便于进行性能分析。

# 不咋的建议

## 不要为了重用而重用 .

存储Bean的定义和协议Bean的定义最好不要为了重用而特意定义成一份当存储和协议的结构【确实】一样时，允许使用同一个Bean定义。一般情况下，两者最好使用独立的Bean定义。建议协议可以包含存储Bean，但不要反过来使用。

## 尽量把逻辑服务器设计成无状态的

尽量把所有的数据都定义到数据库中，让zeze管理所有数据。即使是仅用于一个逻辑服务器的数据也可以存储到后台数据库。这带来的好处是数据量不会局限于内存容量（因为cache会自动管理），后台数据库可以看成容量几乎无限；热更新服务器等事情也会变得简单；请求派发（通过linkd）也会简单。

## 定时器：Timeout & Timer

定时器本质上是程序执行状态，这个破坏了逻辑服务器的无状态性。在大规模集群的情况下，定时器的实现需要解决两个问题：

* 一个定时器应该只在一个逻辑服务器检测。多个检测实例在最终处理事务时忽略重复是可行，但存在浪费。
* 调度定时器的逻辑服务器死掉以后，怎么发现和重新分配到其他逻辑服务器上。

一般建议：

* 定时器数据持久化时建议存储到期（或者下一次时间）的绝对时间，不要存储1小时这种相对时间。这种方式可以利用底层已有的调度注册一次，不用轮询。
* 定时器实际调度时尽量应该和在线用户相关，不要在全部用户上实现。也就是说用户上线时，把他需要的定时器进行调度，此时可以判断绝对时间，把已经超时的处理掉。
* 某些情况下定时器可以交给客户端实现，服务器只校验。比如对于现时使用（不是装备也不会影响buf）的物品，服务器就可以不做实际调度，仅在使用物品的时候检查绝对到期时间。如果用户作弊或者时间不一致，以服务器为准。
* 对于系统级别的定时器，最好都不要轮询。这种定时器看需求了，真的很多，会有相当的负载。实现时尽量注意。

需求分类分析：

* 对于用户相关定时器。用户登录时选定一台服务器，所有相关定时器都注册到这台服务器。如果这台服务器关闭，用户需要重新登录并选择新的一台服务器，再次进行相关注册。
* 对于系统级别（比如定时活动）的定时器。所有的服务器都同时进行定时器判断，忽略服务器之间的时间差，这个问题不大。考虑到不间断运行，这种定时器如果从配置中读取，最好支持运行期重新加载。可以考虑把这种定时器配置存到后台数据库。
* 其他类别。具体问题具体分析了。
* 综上所述，对于Timer，建议使用第三方系统。

## 尽可能使用Rpc

## Events 的建议

1. 如果事件需要和当前事务一起提交回滚: 直接调用实现者的方法。这种模式不建议使用动态订阅的模式，最好就不使用管理类，直接把需要执行的Handle调用写在触发点。这样能很直观的看出来总共有哪些handle。
2. 如果事件需要和当前事务一起提交，但是event-handle允许失败：使用嵌套事务方式执行。
3. 如果事件和当前事务没有直接关联（或者仅仅传递一下参数）：只用Zeze.Util.Task.Run执行。传递参数的时候注意不能把Table内的Bean的引用（beankey的引用可以）直接传过去。一般来说这种情况是不必要的，因为新的事务可以直接查询，此时只需要传递Table.Key即可。
4. events派发的时候一个handle失败是否影响其他handle的派发。一般建议不要扩大影响，也就是说每个handle派发采用同步调用（直接invoke）需要try。Zeze.Util.Task.Run 已经处理了错误，所以就是独立，不影响的。

## 事务的划分

这是并发编程里面最根本也最重要的问题。

1. 最基本的划分规则应该根据需求来决定操作是否放在一个事务中。
2. 一般框架中有Event的模式，此时要注意Event的执行是否需嵌套在触发的事务中执行。建议是除非这是需求决定的（参见上一条），应该启动Event派发放到另外的事务中。

虽然说事务的划分应该根据需求来定，但很多时候，提需求的人也不一定说得清。服务器都是收到一个请求开始处理数据，这样每个请求的处理就可以看作一个事务。这样就不用费太多脑细胞去考虑划分的问题。但是要下面几个例子要注意。

1. 当请求需要对队伍中的多个用户发放奖励时，一般来说一个用户的奖励发放失败不应该影响其他用户。此时就需要分到多个事务，或者使用嵌套事务。
2. 当对所有好友（家族成员，帮派成员）进行遍历处理时，也要注意。总的来说就是需要遍历（广播）之类的操作都需要注意一下。比如Events的触发很多时候也是一种遍历(广播)。
3. 并发优化：如果请求操作很复杂，访问很多数据，冲突比较严重的时候，需要仔细考虑需求，进行事务划分细化。
4. 广播需要一个事务例子：创建角色的时候，需要调用所有需要初始化的模块进行处理，此时不管需要初始多少个模块，都需要在一个事务内。

## TableCache.Capacity

一个事务中操作的同一个表的记录数超过Capacity，这个事务处理时间又超出CacheNewLruHotPeriod时间（超过后就可能会被Clean），这导致事务最后lock\_and\_check时发现记录已经被Clean，然后重做，最终有可能永远完成不了。所以！注意！这个问题可以在记录内记录一个标记，保证新装载的记录至少用过一次才会被Clean，保证至少完成一次事务。但为了这个问题做这些修改，感觉不值得，就不考虑了。

## AllowDirtyWhenAllRead SelectCopy SelectDirty

1. AllowDirtyWhenAllRead 当事务中所有的操作都是读操作并且事务级别为这个，那么事务将不进行原子性检查，直接成功，不会发生重做。具有很高的并发性。非原子性的例子：事务Writer修改两个变量V1,V2（最简单的，来自同一个Bean的两个变量，包括来自两个记录或者来自两个表）；事务DirtyRead先读取V1，再读取V2，那么读到的V1可能事务Writer修改前的，读到的V2可能是事务Writer修改后的；也就是说两个变量没有原子化。这个级别一般用于用户仅仅查询数据用来显示，并不关心数据之间的原子性，也没有关联两个变量的逻辑判断，可以大大提高并发性。
2. SelectCopy 在记录读锁内获得记录的拷贝，如果上面例子的两个V1,V2都在一个记录内，那么原子性得到保证。但是V1,V2在两个记录内（或者两个表），仍然没有原子保证。这个方法可以在事务外使用。Zezex/Game/Login/Onlines 给在线用户发送消息时，可以使用这个方法安全的在事务外执行，因为Online.Status需要的两个变量(LinkName,LinkSid)都在一个记录内，不会发生读到一个修改后的LinkName，而LinkSid又是旧的问题。
3. SelectDirty 一般用于事务外，直接返回数据引用，记录锁外直接读取数据。和AllowDirtyWhenAllRead一样，没有原子性保证。
4. AllowDirtyWhenAllRead SelectDirty 使用时，读取record.var以后，再次读取，值可能发生了变化，所以对同一个var，最好仅读取一次。当然也为了效率考虑，一个变量如果后面需要重用，自己先保存一下。AllowDirtyWhenAllRead 现在除了可以在NewProcedure时设置，也配置到了Protocol中，可以后期优化并发时加上配置，先不用关心。【注意：最好保持同一个变量仅读取一次】

## 缓存同步 & CheckpointMode.Table

由于技术原因，启用缓存同步不能使用CheckpointMode.Period。

## CheckpointMode.Table 的并行优化

在这个模式下，多个事务访问的记录当存在交叉时，会被关联到一起进行Checkpoint。关联越分散，并行度越高。按自然的方式划分事务，一般具有足够好的并行度。但还是需要注意某些全局模块访问。比如有个全局统计数据，非常多的事务都需要读写这个数据进行逻辑判断，那么这些事务都会被关联起来，降低Checkpoint并行度。这时候，提高并行度就需要进行额外处理。根据自己的事务划分需求，看看是否能把读写分到另外的事务中执行，本事务根据结果进行处理。Checkpoint并行度属于优化，开发初期可以不用关心。

## Disable Schemas.Check When Table Is New

新开发的模块可能变动很大，很容易出现不兼容的结构定义。目前一般操作是清除数据库。对那些比较稳定的模块来说，也清除数据，挺影响测试的。所以提供了这个功能：当启动的时候发现Table是新创建的（原来不存在），并且配置了（zeze.xml:DonotCheckSchemasWhenTableIsNew=true)，就忽略这张表的结构兼容性判断。有了这个功能，就按下面的步骤只清除不兼容模块的表。

1. 关闭所有服务进程。
2. 使用后台数据库相关命令删除不兼容模块的表。
3. 启动服务。
4. 自己删除表还是比较麻烦的，发现不兼容的结构自动删除相关表有点风险，看看再提供，由于这个肯定仅用于开发期，考虑仅在MySql上实现。

## 事务中不能访问大量的记录

一个事务不能修改太多的记录；一个事务不能读取太多的记录（比写可以多些）。当循环处理的时候，要注意循环的量级。并且需要注意这个循环是否一定需要在一个事务内。参见事务划分。

## 服务器开发读写问题

一个操作会引起大批量的修改事务时要非常注意，出现这种情况时，看看操作流程能不能转换成少量的写+大量的读取。比如微博的博主发布了一条消息，马上推送消息到关注者的队列中就可能会引起大量的修改事务，改成查询关注对象是否有更新并拉取，这样写只需要写自己的发布队列，问题就转换成大量的查询。因为查询是读，可以充分利用上cache机制，从整体来看是比大量修改事务好的。当然具体问题具体分析，可能不是所有的大量写问题都可以做这个转换。开发同学需要自己注意这个问题。

# Performance

## Cache命中率

Zeze 直接使用本进程内的Cache。在Cache命中的情况下，没有任何远程访问。此时性能可以达到最高。Zeze 的性能核心就是Cache命中率。负载分配的一个原则就是需要提高Cache命中率。

## 记录大小

Zeze 使用后台 key-value 数据库保存数据。记录读取和写入是作为整体保存到后台数据库的。如果记录太大，只修改里面的少量数据，也需要整个记录一起保存到后台。这里有一定的系列化开销。需要分析需求选择合适的记录大小。一般来说应用得到需求都是按模块给的，开始的时候数据按模块划分即可。模块太大时最好分成子模块，或者模块内分成多个小一点的记录。

记录可以包含容器，一般需要设定合适上限。如果数据需要很大，那么应用可能需要自己在key-value记录的基础上实现list（多个记录来保存数据），然后自己实现遍历，增删等操作。

## 缓存同步和分布式事务

这个是全球同服的基础。当需要根据用户量增长不停增加服务器时，可能都有个疑问：吞吐量能提高吗？如果全部的请求都要求互斥的访问同一个数据，那么系统吞吐量怎么弄都是是上不去的。我相信这个世界是天然并发的。一般来说用户请求都访问自己的数据（局部数据）。多个请求是可以并发的。由于Zeze的分布式事务还没有大规模全球同服的实际应用，虽然Zeze的设计上没有限制规模，但有可能会碰到分布式瓶颈。目前考虑到的瓶颈主要是全局单点并且无法并发的模块（这个模块的操作都访问同一个数据）。

## 全局模块的并发

全球同服的系统里面有些模块可能是全局的。这些请求都访问同一个数据，肯定是互斥申请锁排队执行。最高性能就是单个线程全速运行的事务数，这是有上限的。随着用户增长，请求量可能超过上限。此时需要采取一些方案提高数据的并发度。这种全局模块需要提高并发，常见的方式是按某种规则把数据分成多个部分，参考ConcurrentHashMap的实现。通常的数据分块方式例子如下：

1. 记录数据很大，可以分成小块。这样就直接提高了并发。
2. 把数据分成多份（如果可以）。比如某个公司账号有大量的并发转账请求，此时可以建多个子账号。转入操作根据转入者Id的Hash选择某个子账号，这样转入就并发了。转出操作也按这个规则找到开始的子账号。由于该子账号可能金额不够，这是按顺序继续扣后面的子账号。此时转出访问了多个记录，这是没问题的。但是多数情况应该只需要访问一个子账号，不够的情况肯定是少的。读取操作可以分别显示子账号，或者统计一下。读取会导致执行转入账号的服务器的Cache降级到Share。读请求很多的时候，可以用一个定时更新的cache减少实际的数据访问量。
3. 上面Arch部分提到的RedirectHash就是一种把数据分组的规则，也是Zeze目前提供的一种策略。乐观估计，RedirectHash可以解决相当一部分全局单点模块并发问题。

## 高并发设计应该按需行动

如果可以预见请求量，并且代价不大时，可以一开始就优化并发性能。否则可以等到请求量大到快无法支撑了再来优化。一开始实现一个支持任意请求量是没有必要的。计算机都是在有限资源有限时间解决问题。

## Benchmark

|  |  |
| --- | --- |
| Java Bechmark | Result |
| 单线程顺序事务 | tasks/s=1495740.77 time=6.69s cpu=8.27s concurrent=1.24  ZezeJavaTest::ABasicSimpleAddOneThread.java 循环执行存储过程估计被java强烈优化，所以这个数值有点高。 |
| 多线程并发强冲突事务 | tasks/s=252613.37 time=3.96s cpu=9.84s concurrent=2.49  ZezeJavaTest::BBasicSimpleAddConcurrentWithConflict.java 强烈冲突意味着事务几乎总是重做，由于乐观锁重做时保持锁定状态，只会重做一次，所以concurrent为2.49符合预期。 |
| 多线程并发一般冲突事务 | tasks/s=1140652.44 time=4.38s cpu=15.58s concurrent=3.55  ZezeJavaTest::CBasicSimpleAddConcurrent.java |
| GlobalAsync | >50w/s |
| Global虚拟线程 | >15w/s 当前锁不匹配虚拟线程，否则应该接近Async |
| GlobalWithRaft虚拟线程 | 5w/s |
|  |  |
|  |  |

# 配置和自定义配置

配置由Zeze.Config类管理。

## 简单装载

Zeze.Config.Load(String filename);

Zeze.Config.Load() 使用fileName=“zeze.xml”装载配置。

## 自定义配置

1. 定义自己的配置类

public class MyConfig implements Zeze.Config.ICustomize

实现方法 getName，Parse。

1. 装载

Var myConfig = new MyConfig();

Var config = new Zeze.Config().AddCustomize(myConfig);

config.LoadAndParse(filename); // 默认使用“zeze.xml”

自定义配置需要在Load前加入自定义配置类的实例，所以需要先创建Config实例，不能使用简单装载（静态方法）。把myConfig保存到某个全局单例类中，以后直接访问myConfig即可。

## Zeze配置列表

TODO

# Inside Zeze

## 协议基础

zeze框架所用的协议是为TCP payload的流特性而设计的，一条协议紧接下一条协议在流中传输。每条协议的头部结构都能确定这条协议的完整长度，这样就能从数据流中方便地先取出一条完整的协议数据再去进一步解析其中的内容。为了方便在解析内容前确定解析者，协议头部还需类型字段。因此一条协议的序列化结构设计如下(见Zeze.Net.Protocol::Decode)：

uint32 moduleId // 模块ID

uint32 protocolId // 协议ID

uint32 size // 协议数据长度

byte[size] data // 协议数据

其中uint32的序列化是小端字节序，后面提到的固定长度整数和浮点数类型均指小端字节序。

模块ID = 0用于定义框架内部协议，而具体应用应该从1开始定义。每个模块都有完整的32位整数范围的协议ID可用。

协议类型是由高32位的moduleId和低32位的protocolId组合成64位的整数。

框架根据协议类型找到注册过的协议处理器(Zeze.Net.Service.ProtocolFactoryHandle)，处理器包括协议对象的构造工厂、处理方法和事务类型。处理时先用构造工厂构造出对象，再用协议数据调用对象的Decode方法反序列化出协议内的各个成员，最后根据事务类型选择合适的方式调用协议处理方法并传入协议对象。

除了普通协议(Zeze.Net.Protocol)，还有一类协议叫RPC(Zeze.Net.Rpc)。普通协议只包含一个bean类(Argument)；而RPC包含两个bean类(Argument和Result)。普通协议只能单向请求，而RPC能往返发送请求(Argument)和回复(Result)。

普通协议的序列化结构如下：

varint ResultCode // 协议处理结果状态(其实对普通协议来说可能没什么用)

Bean Argument // 协议请求的bean对象(包含各请求字段)

RPC的序列化结构如下：

bool IsRequest // true:请求; false:回复

varint SessionId // 请求和回复的SessionId相同，用于回复时找到对应的请求，由发送方确保唯一

varint ResultCode // 协议处理结果状态

Bean ArgumentOrResult // 根据IsRequest决定序列化Argument还是Result

还有一类特殊的RPC叫RaftRpc(Zeze.Raft.RaftRpc)，其序列化结构在RPC的ArgumentOrResult前插入：

UniqueRequestId Unique // 唯一的请求ID。用于确定重发时去重，而SessionId在重发时仍然会更新

string ClientId // 客户端标识

varint RequestId // 唯一的请求ID的整数值

varint CreateTime // RaftRpc的创建时间，重发时不会改变

模块和协议在一个或若干个xml配置文件中手动定义，由框架的Gen工具生成模块类、协议/RPC类、bean类等代码。其中bean类包含各字段的序列化(Encode/Decode)方法，各种数据类型(包括上面提到的varint、bool和string)的序列化规则详见框架中的文档(doc/zeze序列化标准定义.txt)。

无论是客户端和服务器之间的协议还是服务器之间的协议都按照以上规则传输请求和回复，如果需要对协议加密或压缩，则应在协议序列化之后、反序列化之前转换数据。

在框架的封装下，发送协议只需构造协议对象并调用其Send方法即可异步发送；处理协议只需在生成的模块类里找到生成的协议处理空方法，在里面编写对协议对象参数的处理。如果发送RPC请求，可调用SendForWait获取Future并同步等待，或者调用带responseHandle的Send方法异步处理回复。接收RPC请求并处理后必须调用该RPC的SendResult方法回复到请求方。请求方收到RPC回复时，框架会根据对应的请求RPC中有Future则SetResult打断同步等待，有ResponseHandle则调用其方法处理回复。

发送RPC请求时，框架会生成新的SessionId并记录该RPC，以备接收该RPC的回复时找到对应的请求RPC。收到回复时框架会给对应的请求RPC赋上Result bean再进一步处理。每个RPC都有指定的超时时间，如果请求方发现某RPC超时则设置其ResultCode为Zeze.Transaction.Procedure.Timeout再处理其Future或ResponseHandle。之后如果再收到该RPC的回复则因无法找到对应SessionId的请求RPC而在日志中报错并忽略其处理。

## 客户端登录流程

1. **网络连接**
2. linkd:Zezex.LinkdService(默认11000端口) <= 客户端 (只有此连接配置了加密和压缩)
3. linkd:Zezex.ProviderService(默认21000端口) <= server:Zeze.Arch.ProviderService
4. ServiceManagerServer(默认5001端口) <= linkd:Zeze.Services.ServiceManager.AgentClient
5. ServiceManagerServer(默认5001端口) <= server:Zeze.Services.ServiceManager.AgentClient
6. Zeze.Services.GlobalCacheManagerAsyncServer$ServerService(默认5555端口) 对于单server暂时没有用
7. 以上网络连接均使用下面的通用握手协议和流程。
8. **通用的握手协议和流程(Zeze.Services.Handshake\*)**
9. TCP握手成功后，服务端先发送SHandShake0给客户端，其中唯一的参数(bool EnableEncrypt)表示是否启用加密(可配置)。不过旧版框架没有这个协议，由客户端自己决定是否加密(应该跟服务器保持一致的配置)。
10. 客户端收到SHandShake0后，如果EnableEncrypt=false，则跳到下面的步骤(6)。
11. 客户端发送CHandshake给服务器，其中dh\_group默认为1(可配置)，dh\_data是2^本地生成的随机大整数与dh\_group指定大整数常量的模(详见Zeze.Services.Handshake.Helper)。
12. 服务端收到CHandshake后，计算dh\_data^本地生成的随机大整数与dh\_group指定大整数常量的模，经过HmacMd5的结果作为加密key，加密算法为"AES/ECB/NoPadding"，默认同时启用双向压缩(可配置)。先启用网络输入的加密压缩，然后发送SHandshake给客户端，其中dh\_data是2^本地生成的随机大整数与dh\_group指定大整数常量的模，s2cNeedCompress和c2sNeedCompress表示是否启用双向压缩(均可配置)，发送后即开启网络输出的压缩加密(详见Zeze.Services.HandshakeBase)。
13. 客户端收到SHandshake后，同服务器的算法计算出一致的加密key并按协议参数中的s2cNeedCompress和c2sNeedCompress确定是否启用双向压缩。
14. 客户端发送CHandshakeDone给服务器，并回调Service的OnHandshakeDone通知协议层握手成功。
15. 服务端根据配置和当前的加密状态做验证，成功后回调Service的OnHandshakeDone通知协议层握手成功。
16. **客户端应用层登录流程**
17. 以上的协议层握手成功后，客户端给linkd发送认证请求(RPC:Zezex.Linkd.Auth)，请求包含账号名及token。
18. linkd认证成功后，回复客户端成功(通过RPC回复ResultCode，目前总是成功，没有失败流程)。
19. 客户端确认认证成功后，开始通过linkd跟server通信(通过Dispatch/Send封装协议)
20. 客户端先向server发送角色列表请求。
21. server回复到客户端角色列表后，客户端判断如果角色列表为空，则继续向server请求创建角色，否则跳到步骤(7)。
22. server创建角色后回复客户端成功，失败则通过指定的错误ResultCode回复。
23. 客户端选择已有角色后，向server发送登录角色请求，请求包含登录的角色ID。
24. server确认登录角色后回复客户端成功，失败则通过指定的错误ResultCode回复。
25. 后续客户端还会跟server继续主动获取角色信息及进入世界等协议/RPC。

## 客户端与server的协议包装

1. linkd的LinkdService在收到linkd未知的协议时会封装成Zeze.Builtin.Provider.Dispatch协议，根据未知协议的模块ID找到所属的server服务器并发送给它，如果找不到则回复客户端Zeze.Builtin.LinkdBase.ReportError然后关闭连接。
2. server收到Dispatch协议后，根据其中的协议类型找到ProtocolFactoryHandle，创建协议对象、反序列化并根据事务执行方式调用内部协议的处理方法，其中协议的UserState记录了所属的linkd及其连接客户端的linksid、账号名等上下文。
3. server通过linkd向客户端发协议(包括回复RPC)时，需要把协议封装成Zeze.Builtin.Provider.Send协议并发送给linkd，linkd在处理Send协议时根据其中的linksid向客户端发送内部协议数据。一个Send协议支持附带多个linksid实现小范围广播。
4. server需要全服广播协议时，把该协议封装成Zeze.Builtin.Provider.Broadcast协议，然后广播给所有连接的linkd。

## 客户端登出流程

1. 客户端向服务器发送登出请求(Zeze.Builtin.Online.Logout)。
2. 如果服务器找不到上下文所属的角色ID和登录状态则回复失败；否则向所属linkd发送Zeze.Builtin.Provider.SetUserState协议清除linkd的状态，再回复Logout成功并清除登录状态，然后让各模块处理角色登出。
3. 客户端收到Logout回复成功后，主动断开网络连接。

## 服务器之间的协议

1. 服务器可以有多个linkd和多个server。其中所有的linkd跟所有的server都有连接(通过ServiceManager注册和订阅)。而linkd之间没有连接，server之间有完全连接(总是小的serverId主动连接大的serverId)。
2. 服务器之间的连接通常不配置加密和压缩，协议和RPC直接发送无需包装。
3. linkd注册的服务信息：服务名:"Game.Linkd"; 服务ID:"@{providerIp}:{providerPort}"。
4. server注册的服务信息**:** 服务名:"Game.Server.Module#{moduleId}"; 服务ID:"{serverId}"。
5. server订阅"Game.Linkd"服务名，得到所有linkd的provider的IP和端口并主动连接，server在协议层握手成功后主动给各linkd依次发送Zeze.Builtin.Provider.AnnounceProviderInfo协议、RPC:Zeze.Builtin.Provider.Bind(静态模块)和RPC:Zeze.Builtin.Provider.Subscribe(动态模块)。
6. server订阅"Game.Server.Module#{moduleId}"各模块的服务名，得到所有server的providerDirect的IP和端口并主动连接比自己serverId大的server，在协议层握手成功后主动连接方给对方发送Zeze.Builtin.Provider.AnnounceProviderInfo协议。

## 协议的字段兼容性

1. 所有的数值类型(byte,short,int,long,float,double)之间互相兼容(运行时透明自动转换)，转换规则跟所用编程语言的强转相同。所以注意高位截断等问题，通常应该从小范围类型改成大范围类型。
2. bool类型跟数值类型也互相兼容，bool转数值成0和1，数值转bool会用"!=0"来取结果。
3. binary和string类型互相兼容，string的序列化是用UTF-8编码成binary，binary转string时如果遇到无法UTF-8解码时会抛异常。Lua的string因为无需编解码所以跟binary完全兼容。
4. bean类型的兼容性只看字段ID和字段类型，与bean的命名和类型ID无关。序列化数据里缺失的字段会当成默认值(0,false,空binary,空string,空容器,所有字段均为默认值的bean,内容是EmptyBean的DynamicBean)。如果序列化数据里有当前bean类型中不存在的字段ID，则直接忽略，再序列化时就会丢弃该字段。
5. dynamic类型只是附带类型ID而不是根据配置选择具体类型名的bean，根据配置指定的方法用类型ID取得具体的bean类型再反序列化。如果改成bean类型，则忽略类型ID直接按指定的具体bean类型反序列化。如果从bean类型改成dynamic类型，则用默认0当作dynamic的类型ID。
6. 序列容器(list,set,array)互相兼容，容器内类型的兼容性同上。
7. 关联容器(map)内key和value的类型兼容性同上。
8. 以上没提到兼容性和转换规则的转换均不支持，遇到则用默认值取代旧值。
9. 更多相关规则详见框架中的文档(doc/zeze序列化标准定义.txt)
10. 关于bean增减字段的建议：按字段ID的顺序从1开始自增地分配和扩展；删除字段不要直接删除，可以修改字段名或注释来表示“临时不再使用”的含义，方便保留数据库中已有的数据不丢失，以备之后再恢复使用，也防止增加字段时重用该字段ID引发取出旧数据的混乱。如果有彻底全服删库的机会，可以删除不会再用的字段，此时也可以顺便重新整理所有的字段ID。

## ModuleRedirect流程

1. App类在创建模块时会调用ReplaceModuleInstance方法，它发现模块类中有修饰了以下几种注解的转发方法就会使用转发类替代原模块类，它会先判断当前类路径中是否已经存在转发类，如果没有再为此模块生成转发类代码；生成后不会自动生成文件，而是配置了JVM属性"GenFileSrcRoot=输出路径"才会输出到文件，命名为"Redirect\_模块全名.java"，其中"模块全名"中的点号转成了下划线。生成的类继承原模块类并重写所有转发方法，其构造方法会注册所有转发方法全名(包名:方法名)及对应的处理方法。通常App类在创建并Replace所有模块后，判断配置了GenFileSrcRoot就直接退出，用于只生成代码而不启动服务。除RedirectAll外其它各转发类型的转发流程大同小异。一共三种转发类型如下：
2. Zeze.Arch.**RedirectToServer**：单发给首个参数"int serverId"的模块执行该方法并回复结果，找不到serverId会抛异常，serverId也可以是当前的server，此时会自动同步调用原有的方法。返回类型可以是void或Zeze.Arch.RedirectFuture<自定义结果类型或Long(resultCode)>。自定义结果类型必须是public的，传输结果的字段也必须是public的且支持普通值类型或可序列化(Zeze.Serialize.Serializable)的类型，请求方法的其它参数也有此要求。自定义结果类型可选包含特定的字段"long resultCode"用于接收执行结果的状态。执行者的server在收到转发请求(Zeze.Builtin.ProviderDirect.ModuleRedirect)时会运行模块原有的方法代码，通过RedirectFuture.finish传入自定义结果类型或Long(resultCode)回复请求者的server，可异步回复，方法返回不会自动回复。请求者的server可对返回的RedirectFuture调用await来同步等待结果，或者调用then传入回调来异步处理结果，这两个方法可连接使用，如"...(...).then(...).await()"。获取到结果应先判断resultCode是否超时。
3. Zeze.Arch.**RedirectHash**：类似RedirectToServer，只是首个参数换成"int hash"，根据hash值选取一个提供了该模块的server。其中该注解可附带一个ConcurrentLevelSource字符串参数，表示获取并发级别的来源代码，该代码会嵌入到生成代码中，并发级别默认是1。如果并发级别<=1，则选取server的规则是在提供了该模块的服务列表中以一致性hash算法(见Zeze.Util.ConsistentHash)选取一个server；如果并发级别>1，则先对hash以并发级别取余并hash后再用一致性hash算法选取一个server。
4. Zeze.Arch.**RedirectAll**：用于广播给多个提供该模块的server执行并收集所有结果。要求首个参数必须是"int hash"，请求发起者传入该参数表示并发级别，并发级别多于server总数时某些server需要处理多次，各执行者可从hash参数得到[0，hash-1]中某一值。方法的返回值必须是RedirectAllFuture<继承RedirectResult的自定义类型>，自定义结果类型的要求类似RedirectToServer，但不必定义resultCode可以从其父类RedirectResult中获取hash和resultCode。执行者的server运行原方法时返回"RedirectAllFuture.result(传入自定义结果类型)"来同步回复结果，如需异步则返回"RedirectAllFuture.<自定义结果类型>async()"，其返回对象在其它方法中调用其"asyncResult(传入自定义结果类型)"方法来完成回复。请求者的server可对返回的RedirectAllFuture调用await来同步等待结果，调用onResult来处理每个结果，调用onAllDone来得到RedirectAllContext对象，可获得所有结果。这三个方法可连接使用，如"...(...).onResult(...).await().onAllDone(...)"。获取到每个结果都应先判断resultCode是否超时。
5. 以上各种转发方式的用法示例见框架项目中的Game.Rank.ModuleRank。
6. ModuleRedirect协议不通过linkd转发，而是在启动时通过ServiceManager注册和订阅，获得所有server提供的模块信息，直接在servers之间建立连接并收发ModuleRedirect。

## 多线程基础

使用Zeze几乎不需要了解多线程相关知识，深入理解Zeze则需要掌握一定的多线程知识。Zeze使用了Mutex,Condition,ReadWriteLock,volatile,Future(TaskCompleteSource)。多线程有系统提供的同步原语可用，可以分析公共数据是否都具有合适的保护，执行流程是否死锁。普通多线程还是可以进行有效分析的，相对异步来说要简单点。一般应用内死锁自己都可以避免，但是框架有时需要在锁内执行用户操作，这很可能带来死锁风险。异步问题一般都通过状态处理，比如Zeze/Raft/就是异步的，这个更难分析。多线程、异步问题的本质一样的，这里有个我叫做“时间窗口敏感性“的东西，也就是说你能主动察觉到某些地方可能有问题，需要通过加同步原语保护或者加状态检测。《**UNIX网络编程 卷2 进程间通信 第2版**》旧版里面的信号安全的处理（由于信号这个东西基本没人用了，新版不知道有没有被删除），可以锻炼自己的时间窗口敏感性。最后举个异步的例子：

系统有Login，Logout两个操作，当Logout的执行滞后了，在新的Login之后才执行，那它就会出问题，这个问题无法用锁（也不能在事务内保护）简单解决，一个可行的处理方案是使用Login系列号，Logout时判断当前的Login状态（系列号）根自己的不一样，忽略这个Logout的执行。

## 事务怎么实现回滚

修改数据时在当前事务中记录日志，在事务提交时，提交日志。Zeze只在Transaction中提供GetLog，PutLog两个接口，另外为不同的修改提供了几个日志类。日志的访问和创建都在Bean的生成代码中实现。下面是一个简单的Bean的生成代码例子（不完整）：

public final class Simple extends Zeze.Transaction.Bean {

private int \_int1; // com aa

public int getInt1() {

if (!isManaged())

return \_int1;

var txn = Zeze.Transaction.Transaction.getCurrent();

if (txn == null)

return \_int1;

txn.VerifyRecordAccessed(this, true);

var log = (Log\_\_int1)txn.GetLog(this.getObjectId() + 1);

return log != null ? log.Value : \_int1;

}

public void setInt1(int value) {

if (!isManaged()) {

\_int1 = value;

return;

}

var txn = Zeze.Transaction.Transaction.getCurrent();

assert txn != null;

txn.VerifyRecordAccessed(this);

txn.PutLog(new Log\_\_int1(this, 1, value));

}

private static final class Log\_\_int1 extends Zeze.Transaction.Logs.LogInt {

public Log\_\_int1(Simple bean, int varId, int value) { super(bean, varId, value); }

@Override

public void Commit() { ((Simple)getBelong()).\_int1 = Value; }

}

}

完成的类型和容器的日志使用例子可以到zeze\ZezeJava\ZezeJavaTest下运行gen.bat，并查看zeze\ZezeJava\ZezeJavaTest\Gen\demo\Module1\Value.java

## 事务和乐观锁

存储过程执行过程中不加锁，所有修改仅当前事务可见。提交的时候对所有访问的记录排序并且加锁并进行冲突检查。

### 乐观锁并发原理

事务内所有访问（读写）的记录在冲突检查时需要确保Timestamp没有变化。***事务成功时，相当于独占所访问的记录***。这个并发策略是严格，但显然是正确的。

### 乐观锁算法要点

1. 排序加锁，实际上所访问的记录存储在SortedDictionary中。
2. 加锁后检查冲突，即数据是否改变。冲突则重做事务。
3. 冲突重做时保持已经得到的锁，这样在冲突非常严重时，第二次执行事务一般都能成功，而不会陷入一直冲突，事务永远没法完成的情况。
4. 重做保持锁，但重做过程中所访问的记录可能发生变化，所以重做仍然需要再次执行lockAndCheck逻辑，并且处理所访问的记录发生变更的问题。
5. 逻辑处理返回错误码或者异常时也需要检查lockAndCheck，因为乐观锁在实际处理逻辑时没有加锁，可能存在并发原因导致本来不应该发生逻辑错误，此时的仍然需要加锁并完成冲突检查，如果冲突了，也需要重做。
6. 完整的实现参见 Zeze.Transaction.Transaction. lockAndCheck.\_

## 一致性缓存（分布式事务）

多台服务器共享后台数据库。每台服务器拥有自己的缓存。一致性缓存就是维护多台服务器之间缓存的一致性。记录分成读写两种状态。读状态允许同时存在于多台服务器缓存中。写状态只允许在一台服务器中。zeze一致性缓存和CPU-Cache-Memory的很像。参考了CPU的MESI协议自己实现了一个锁分配机制。仅使用了类似MESI的状态名。***在一致性缓存之上，每一台服务器的事务就能像自己独占所有数据一样，完成本地事务即可。***这就是基于一致性缓存的分布式事务。

### 锁管理流程

参考了CPU缓存同步算法（MESI），使用了其中3个状态：Modify,Share,Invalid。当主逻辑服务器需要访问或修改数据时，向全局锁管理服务器（下面用Global称呼）申请Modify或Share锁。Global知道所有记录的锁的分布状态。它根据申请的锁，向现拥有者发送相应的降级请求；拥有者释放锁，并把数据刷新到后端服务器后，才给Global返回结果；Global登记申请者的锁状态，给申请者返回结果。

### 锁管理算法要点

1. Global在多个记录上并发执行Acquire操作。对单个记录，所有的申请排队互斥，一个一个处理。
2. Global处理Acquire时，除了死锁检测会马上返回失败，多数情况下都会返回成功。
3. 如果拥有锁的逻辑服务器（下面简称Server）没有响应Reduce请求（超时），此时实际发生的情况没法预测，Acquire会失败。
4. 允许Global认为记录权限已经分配个某个Server，但Server实际上没有（比如Server重启了）或拥有较低权限。所以，Server处理Reduce，必须能正确处理Recude的目标状态和自己实际状态，并且返回成功。
5. Server在多个记录上并发处理权限申请；对单个记录，所有的Acquire排队。对同一个记录，不会同时发送Acquire给Global。
6. 实现：Zeze\Services\GlobalCacheManager.cs
   1. AcquireModify,
   2. AcquireShare
   3. Release
7. 实现：Zeze\Transaction\Table.cs
   1. ReduceShare
   2. ReduceInvalid
   3. Load

## 持久化模式

CheckpointMode .Period：定时保存修改到后端数据库，如果保存前进程异常退出，修改会丢失，相当于上一次保存以来的所有事务回滚，数据不会被破坏。这个模式仅用于单机模式，不能和一致性缓存一起使用。

CheckpointMode .Immediately：事务提交的时候马上保存到后端数据库。【废弃，是否恢复以后再说】

CheckpointMode .Table：可以配置某些表是重要的，当事务访问的表有重要的时，这个事务的所有修改会马上保存。事务访问的表全部不是重要的，这些修改定时保存。这个模式适用范围比较广。由于zeze是KV数据库，一个记录即使只变了一部分，也需要整体写入，存在一定浪费（系列化和io成本）。定时保存可以降低后端数据库的写入量。仅把真正重要的表配成马上保存。

主要实现： Zeze/Transaction/Checkpoint.cs Zeze/Transaction/RelativeRecordSet.cs

## TableCache

1. TableCache是Zeze性能的根本，需要配置合适的容量。
2. 事务中删除记录不会马上从TableCache中删除，这样可以实现负Cache。
3. TableCache定时执行清理操作，按Lru规则删除最老的记录。
4. 记录正在被访问、脏的、新鲜的等状态时不会被清理。

# History

## 历史

写程序一开始，我就对检查状态并修改数据感到很困惑。特别是程序复杂分模块以后，此时检查所有的状态，最后修改数据，就需要每个模块状态检查代码提取出来提前一起判断。所以一直希望能有个事务环境，在碰到状态不正确时，回滚所有的修改，把数据 恢复到开始的时候。2007年的时候，开始做游戏，就用java写了个xdb，在程序中支持事务。这个版本在需要访问数据时，马上加锁。由于访问数据的顺序跟逻辑相关，就有可能死锁。当时的解决方法是使用java的死锁检测，发现死锁就打断重做。或者程序员在一个事务 开始的时候把所有需要访问的数据的锁都提前（这是可以排序）锁上。

死锁就成为xdb最大的问题，也是xdb不大好用的地方。

2013年的时候，当时的同事 pirunxi 提出了乐观锁：所有的数据修改先仅在本事务中可见，执行完了以后（此时可以知道所有的数据，就可以排序加锁，就不会死锁了），加锁和判断数据状态，不冲突的话，事务成功，冲突的话保持已有的锁重做。这个方案解决了死锁问题，系统易用性大大提升。

在2014-2017年间，pirunxi 实现了好多个基于乐观锁的版本。

我大概是2015年开始参与讨论。

今年（2020）新冠疫情期间，老婆孩子不在身边，我闲着没事。一次就问了 pirunxi 最新版本的情况。然后（闲着没事）就写了 Zeze 这个版本，

这是我的第一个 c# 程序。正如当时xdb是我的第一个java程序。

## 同一个世界同一个梦想（2008）

不是很喜欢游戏分组的做法。当前很多游戏运营商把不断开新服合并旧服当作一种运营策略，用来增加玩家游戏时间，这种情况下使用可以按单个独立GameServer模式使用Zeze，或者按小规模分布式的模式使用Zeze。当需要全球同服这种运营模式时，就能充分利用Zeze的分布式架构。有了Zeze分布式事务，就能更接近同一个世界同一个梦想。当然由于一些国家的法律法规限制，不能实现全球同服。这个即时我当了联合国秘书长也是解决不了的。也许瓦肯人的出现能解决这个问题。

# Git

[**https://gitee.com/e2wugui/zeze**](https://gitee.com/e2wugui/zeze)

**https://github.com/e2wugui/zeze**

# Q&A

## 使用zeze需要掌握多深的多线程知识？

1. 只访问zeze管理的数据是线程安全的，不需要任何多线程背景。这类问题就是zeze主要解决的，预计能解决相当一部分逻辑开发。
2. 程序内有自定义的共享数据（非Zeze管理的数据），需要使用者自己用线程同步机制保护共享数据。
3. 有自己定义的共享数据并且这个数据和zeze数据有紧密联系。比较麻烦，需要具体问题具体分析。 请参考AutoKey的实现。

## C# & Java

C# 使用了async。Java 是同步多线程。async 版本所需要的线程数量很低（接近cpu-core-number），这降低了线程切换的消耗和降低了内存，资源利用率比较高。Java版在它的虚拟线程发布以后，会考虑马上采用虚拟线程。Java虚拟线程是一个跟async接近，是一个更像普通线程的异步实现。

## 同步、异步接口

同步接口更容易理解，所有的代码流程有顺序，并且都在一起。异步接口会把多个处理流程分散的不同的回调（异步回调）中，维护和理解稍显困难。当某个操作不能阻塞，必须马上接着做其他事情，此时就需要异步接口。建议不是在必要的时候，都是用同步接口。

## Why Not Netty

Netty Is Big.

实际上现在Netty Web已经引入了Netty了。

## NuGet包在“此源中不可用错误“

Vs社区版NuGet包源设置有问题，按下面进行配置。

Name: nuget.org

Source: <https://api.nuget.org/v3/index.json>