# 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10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);

}

# 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（TransacionLevel），然后由Zeze自动创建。

## TransacionLevel

在定义协议的时候，配置TransacionLevel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。

## 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 (!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。

## RunWhileCommit & RunWhileRollback

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。

RunWhileCommit 事务成功提交时执行。

RunWhileRollback 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.RunWhileCommit(() => assert sum == 0);

// 如果没有RunWhileCommit，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。

}

# 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。

## 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 | bool | 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<int> | PMap2<Bean>  PMap1<int> | Table | Map |
| list | CollList2<Bean>  CollList1<int> | PList2<Bean>  PList1<int> | Table | Array |
| set | CollSet1<int> | PSet1<int> | Table | Set |
| dynamic | DynamicBean | DynamicBean | Table | DynamicBean |

# 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：ModuleId+ProtocolId+Size+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。

Rpc提供同步等待，和异步回调接口两种方式处理结果。

## 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。目前用于unity配置。

# 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。

## 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();

# 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

## GlobalCachaManagerWithRaft

# Game

Game是Zeze内建模块。

## Online

## Bag

## 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。现在一般没人用了吧。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

## ProviderImplementWithOnline

# Arch Detail

## Linkd

### LinkdApp

### LinkdProvider

### LinkdProviderService

### LinkdService

## Server

#### ProviderApp

#### ProviderImplement

#### ProviderService

#### ProviderDirect

#### ProviderDirectService

#### ProviderModuleBinds

#### Online

#### LoadConfig

#### RedirectAll

#### RedirectHash

#### RedirectToServer

## 发现Linkd,Server

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。

# 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：

1. 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。
2. 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：

1. 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。
2. 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。c)
3. 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。
4. 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：

1. 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。
2. 对于系统界别（比如定时活动）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。
3. 其他类别。具体问题具体分析了。
4. 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

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。

# 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。

## 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。

## 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。

1. 记录数据很大，可以分成小块。
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。

## TODO全局模块的Cache命中率和ModuleRedirect（在Zezex中）

see Zezex/README.md

see Zezex/linkd.provider.txt

ModuleRedirect 注解简单说明如下：

ModuleRedirect @Redirect

注解参数 ChoiceHashCodeSource

1. 如果没有设置，则按 LoginSession.Account.HashCode 进行重定向。

2. 如果方法需要非Login的环境下用，必须提供 ChoiceHashCodeSource。

ModuleRedirectWithHash @RedirectWithHash

注解没有参数。

方法的第一个参数为 HashCode，由调用者提供。

ModuleRedirectToServer @RedirectToServer

注解没有参数。

方法的第一个参数为 ServerId，这个方法将重定向到该Server。调用者提供。

ModuleRedirectAll @RedirectAll

注解参数 GetConcurrentLevelSource，【必须提供】就是 HashArray.Length

实现将把请求广播到 HashArray 中所有服务器。【根据配置，服务器数量可能小于HashArray.Length，也就是说同一个服务器可能处理多个hash分组】

## 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。

# Inside Zeze

## 多线程基础

# History

# TODO

# Git

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

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

# 发布

# Q&A

雁影:

想知道同步和异步版本的优劣

资源利用率。

雁影:

看到有些功能有两种或者以上实现 想知道他们的优劣或者适用于什么场景

雁影:

3.c#和java实现不同的一些细节

程小三:

记录设计决策，现在方案是这样，当时可选的方案有哪些？为什么选择现有的？

# 肖丽杨:

# 结构设计和技术选型怎么做的？考虑了那些问题呢？网络方面为什么不引入netty而是直接使用nio呢？

因为不熟悉netty；Net实现也没几行实现代码。

雁影:

最好能为一些典型的游戏类型(轻度分服，轻度同服，重度同服，重度分服)各提供一套标准配置案例

李成华:

不知道你这里轻度重度怎么区分的。按原来的框架分为：1，单台gs（没有也不需要一致性缓存），1.1 client 直连，1.2 有linkd，2，小规模同服，多个gs，现在Zeze.Arch支持，3，大规模，结果和小规模一样，但能达到多大规模没有实际案例

李成华:

Zeze.Arch都是有linkd的。单台gs的例子被删除了。

李成华:

直接把gs裸露出来，可能大多数有些结构都不会这样做，一半都有linkd，因为游戏还有其他类型的服务器，需要这个linkd一起代理了。所以单台gs直连实际上不大会发生，我就删除了。

李成华:

“大多数游戏架构都不会”

雁影:

轻重只的同时在线人数

李成华:

单个gs的在线人数？

李成华:

还是整个Zeze.App的同时在线？

雁影:

整个游戏

雁影:

比如我想做个万人在线卡牌游戏该配怎么配置 几个link 几个gs global用不用raft

李成华:

那就是我说的小规模，大规模了，这是从需求来分。zeze集群设计没有最大在线限制，但以后有没有瓶颈还不知道。但对于分服运营来说，小规模肯定能用，这个稍微扩大单个gs在线限制。比如单个gs 3万，集群10台，扩到30万。能满足需求目标的话就没有问题了。

---

万人卡牌，一台gs【直连】我觉得都够了。

性能只能根据基准测试（需要考虑实际应用负载）结果，自己去配。

这个zeze应该给不出建议。

李成华:

我的经验是，mmo，单个gs 2w没问题，卡牌没有经验，但猜 5-10w吧，然后要不要global就看单台是否满足需求，不满足，就global一直括，一直扩到zeze达到某个瓶颈为止。global用不用raft也看需求。

# Basic

## 事务层次图

1. **用户事务层**
2. **Cache**
3. **后台数据库(Mysql,Sqlserver,tikv,...)**

## 用户事务

用户事务通过乐观锁方式并发访问Cache。用户事务提交时把修改更新到Cache中。

乐观锁基础思想

## CheckpointMode.Period

保存的时候暂停事务的执行，把Cache中所有脏数据保存到后台数据库。每次保存在后台数据库的一个事务中。也就是说一次保存包含了上次保存以来的所有用户事务（修改）。

## 思考：用户事务被合并的做法正确吗？

事务被保存到后台数据库中分了两层，从用户事务

## 单一服务器模式

图

如果采用分服方式运营，可以只有一个GameServer后面接后台Database，用户直接连接GameServer，这种模式下，Service Manager和GlobalCacheManager都不需要。

1. 乐观锁基本算法。

a. 事务执行逻辑操作的时候不加锁。

b. 提交前把所有事务内访问的记录【排序】并加锁，检查冲突。

c. 如果【冲突】就重做，否则修改数据并提交。

1. 乐观锁其他要点。

a. 锁住以后，如果冲突，不释放锁重做，一般情况下只会浪费一次执行。

b. 锁住重做仍然需要再次排序【加锁】。需要考虑重做时，锁发生变更，读锁变写锁等情况。

# 核心实现

## 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。

【并发正确性】

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，相当于独占所有访问的记录。这个并发策略是严格，但显然是正确的 。

## 一致性缓存

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。

## 关联记录集合

## 持久化模式

1. **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。
2. **Immediately：** 事务提交的时候马上保存到后端数据库。
3. **Table：** 可以选择部份表，当事务包含这些表时会马上保存，否则定时保存

# 其他特性

* 数据易访问
* 数据完整性严格保护
* 高可用
* 高性能
* 易使用

多线程、异步

多线程有系统提供的同步原语可用，分析的时候：公共数据是否都具有合适的保护，是否死锁。有一定的可操作性，相对异步还是比较容易分析的。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。

多线程

1. 使用zeze需要掌握多线程知识码？

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。

这种使用形式我叫它0类。这类是zeze主要解决的，预期能解决相当一部分逻辑开发。1类就是在程序内定义本进程私有的数据，采用某种多线程模式来访问。由于这是进程内私有数据，zeze管理不到，在多Gs实例的模式下需要自己解决同步问题（如果有）。2类就是有私有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。

# 目标

0) 尽可能减少实现业务逻辑时需要的技术手段。降低应用开发难度。降低成本。

让实现代码离业务逻辑尽可能近，几乎能对照起来。

1) 严格保护数据不损坏。

2) 解决扩容问题，可以很容易构建成千上万台机器集群的服务。

3) 解决高可用性问题，达到7x24小时不间断工作。

4) 简单直接的编程接口，直接融入编程语言。

# 主要特性

0) 事务

提供一般意义上的数据库的事务。

如果操作失败，会把放弃所有修改，把数据恢复到修改前的状态。

a) 异常安全

假设代码中任何地方都可能抛出异常，那么在没有事务支持的环境中

写出正确的代码是非常辛苦的。而现在的语言多数都是会抛出异常。

b) 更容易模块化

跨模块调用前可以先修改数据，调用后发现逻辑条件不符合，可以回滚。

更容易按需求定义模块接口，更容易实现代码尽量接近业务描述。

1) 基于缓存同步的分布式事务

缓存同步参考了CPU的MESI（缓存一致性协议）以后设计的一个简单同步协议。

这个功能由底层实现，应用除了配置不需要访问任何接口。

这是集群扩容的基础。编程的时候，就跟自己独占数据一样。

以后增加运行实例即可达到扩容目的。

2) 简单网络库

真的非常简单，稍作封装后，对一般逻辑开发完全隐藏网络细节。

a) Protocol

b) Rpc

3) 数据库封装

自动持久化到后端数据库。对应用来说，可以当作数据库不存在。

支持的数据库（以后根据需要添加，很容易）。

a) SQLServer

b) Mysql

c) Tikv

4) Raft

用来支持高可靠性。

5) 很容易就能访问所有数据

拥有所有数据，不会碰到到处要数据的问题。

实现逻辑很清晰。

6) 乐观锁

不会死锁。重试会浪费cpu。但一般cpu是富裕的。

7) 高性能（Local Cache）

在本地进程中保存最近访问的记录，当Cache命中的时候，完全没有远程访问开销，具有很高的性能。

#### 实现导读

0) 开始

a) zeze的核心功能就是提供事务，最好开始先了解一下事务的回滚是怎么实现的。

c#可以先看看zeze\UnitTest\gsd\Gen\demo\Module1\Value.cs 需要先生成代码，运行zeze\UnitTest\gen.bat

java可以先看看zeze\ZezeJavaTest\Gen\demo\Module1\Value.java 需要先生成代码，运行zeze\ZezeJavaTestt\gen.bat

b) 下面4个功能的并发是实现的核心。阅读的时候，建议每个功能先分别整理。最后4个部分合起来联调。

下面的功能暂不提供详细的设计方案，这样可以自由发挥，便于查找bug，免得知道了设计方案，会被限制。

1) 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。

核心算法：Zeze/Transaction/Transaction.cs -> \_lock\_and\_check\_()

【并发正确性】

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，相当于独占所有访问的记录。这个并发策略是严格，但显然是正确的。

暂时不考虑实现其他并发级别。

2) 缓存同步（GlobalCacheManager）

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，向全局权限分配服务器（GlobalCacheManager）申请Modify或Share权限。

GlobalCacheManager知道所有记录的权限的分布状态。它根据申请的权限，向现拥有者发送降级请求，

然后给申请者返回合适结果。

核心算法：

Zeze/Services/GlobalCacheManager.cs -> AcquireModify, AcquireShare, Release

Zeze/Transaction/Table.cs -> ReduceShare, ReduceInvalid, FindInCacheOrStorage

当主逻辑服务器收到降级请求时，会把相关记录保存到后端数据库以后才给GlobalCacheManager返回结果。see 下面的持久化模式。

【当前实现规则】

a) GlobalCacheManager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，一个一个处理。

b) GlobalCacheManager处理Acquire时，除了死锁检测会马上返回失败，正常情况下会返回成功。

如果逻辑服务器没有响应Reduce请求（超时），此时实际发生的情况没法预测，Acquire会失败。

see zeze\GlobalCacheManager\Cleanup.txt

c) 允许GlobalCacheManager认为记录权限已经分配个某个逻辑服务器，但逻辑服务器实际上没有（比如逻辑服务器不正常重启了）或拥有较低权限。

所以，逻辑服务器处理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。

d) 逻辑服务器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GlobalCacheManager。

3) 持久化模式（Checkpoint）

a. 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。

b. Immediately 事务提交的时候马上保存到后端数据库。

c. Table 可以选择部份表，当事务包含这些表时会马上保存，否则定时保存。这个模式适用范围比较广。

核心算法：

Zeze/Transaction/Checkpoint.cs

Zeze/Transaction/RelativeRecordSet.cs

4) TableCache.Cleanup

5) 用户事务层-Checkpoint.Flush层，两层事务，整体事务对吗？

#### 安装教程

0) Zeze 是一个类库，所有的核心功能都在这里。

a) 服务器开发建议直接把Zeze\Zeze.csproj加到自己的sln种。

直接引用源码项目。

b) 客户端（unity），建议直接拷贝Zeze下的源代码进项目；

也可以自己编译相应平台的版本，发布到 unity plugins 下。

1) Gen

是一个控制台程序。编译好以后，用来生成代码。

2) GlobalCacheManager

是一个控制台程序。当多个Application共享数据库时，用来管理缓存同步，

3) ServiceManager

管理服务注册和订阅。

让服务器集群更加容易。

# #### 使用说明

【不要捕捉异常】

如果你需要处理错误，捕捉处理完以后，再次抛出。

0) 定义数据结构(Bean)

在xml中定义，使用Gen生成代码。

a) 支持基本类型和容器。

打开文件'zeze\UnitTest\solution.xml' 查找 '<bean name="Value">'。

这个的例子包含了所有的基本类型和几个容器实例。

b) 支持数据结构变更。

可以随时增删变量，而不需要数据库转换操作。

c) Bean.dynamic

用来实现多态类的数据保存。

1) 定义协议(protocol or rpc)

打开文件'zeze\UnitTest\solution.xml' 查找 'protocol' or 'rpc'。

2) 定义数据库表格(table)

打开文件'zeze\UnitTest\solution.xml' 查找 'table'。

3) 定义模块(module)

模块是用来包含Bean,Protocol,Rpc,Table的。

打开文件'zeze\UnitTest\solution.xml' 查找 'module'。

4) 定义项目(project)

一个项目对应一个进程。在这里定义service。

服务器项目目前仅支持c#。

客户端项目目前支持c#，c++；

支持的脚本ts，lua。

打开文件'zeze\UnitTest\solution.xml' 查找 'project'。

5) 数据变更订阅

有限支持数据变更订阅（ChangeListener）。

不管修改数据的操作有多少个，当数据发生变更时，回调Listener。

那么不管有多少个地方（方式）修改该数据，只需要一个地方处理数据变更。

一般用于客户端数据同步。

【注意】除了数据同步，不建议在业务逻辑实现中使用这个特性。

6) 支持后端多数据库

用来避免后端数据库性能不足。

由于Zeze的性能来源主要来自本地cache的命中，一般不会发生后端数据库性能不足。

7) 网络连接支持加密压缩。

使用 Diffie–Hellman key exchange 交换密匙。

8) 配置（zeze.xml）

参考：Game\zeze.xml; UnitTest\zeze.xml;Game2\zeze.xml

一般来说，开始需要提供一个数据库配置，其他都可以用默认的。

不提供配置的话，数据库是内存的。

9) 什么时候创建存储过程（Zeze.NewProcedure）

现在框架默认为每个协议创建存储过程，一般来说不再需要自己创建。

如果你想要事务部分失败的时候不回滚整个事务，那就需要嵌套事务，

此时需要创建自己的存储过程并判断执行结果。

int nestProcedureResult = Zeze.NewProcedure(myaction, "myactionname").Call();

// check nestProcedureResult

10) Bean More

纯粹的数据对象，里面可以包含Bean，容器。容器里面又可以包含Bean）

【reference】

所有的 bean 引用不允许重复，不允许有环（TODO Gen的时候检测环）。

【null】

所有的 bean 引用不会为 null，使用的时候不需要判断，可以简化代码。

【Assign & Copy】

赋值和拷贝。

【Managed】

Bean被加入Table或者被加入一个已经Managed状态的Bean的容器中之前是非Managed状态，

此时修改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，参考上面的reference说明。如果你想加入重复的对象，

使用 Bean.Copy 方法复制一份。

【binary】

Zeze.Net.Binary，创建以后不能修改，只能整个替换。

【dynamic】

Bean的变量可以是动态的，可以在里面保存不同的bean。

see Game\solutions.xml：Game.Bag.BItem的定义。

dynamic 在声明支持的Bean类型时，可以指定所属Bean的范围内唯一的TypeId。

这个Id会被持久化。如果没有指定TypeId，则默认使用 Bean.TypeId。

【Bean.TypeId】

Bean.TypeId 默认使用Zeze.Transaction.Bean.Hash64(Bean.FullName)生成。

冲突时，可以手动指定一个。

Bean的FullName发生变化，而又想兼容旧的数据时，需要手动把TypeId设置成旧的hash值。

11) Protocol.id Rpc.id

默认使用Zeze.Transaction.Bean.Hash16(Protocol.FullName)生成。

冲突时，需要手动指定一个。

12) Session & UserState

AsyncSocket.UserState

连接上下文

Protocol.UserState

从某个连接收到的所有协议的上下文，默认从连接上下文复制引用。

Procedure.UserState

当为协议处理创建存储过程执行事务时，默认从协议上下文复制引用。

see Game\game.sln, Game2\game2.sln

13) 日志和统计

记录了几乎所有的错误日志。

统计了几乎所有各种可能的情况。（可以通过宏完全关闭）。

Zeze 记录日志的时候会把 UserState.ToString 也记录进去。

应用可以在自己的UserState对象实现类中添加更多上下文信息。

比如, Login.Session.SetLastError("detail");

这样写的时候只需要返回错误，不用每个地方自己记录日志。

14) 协议和存储过程处理结果返回值规划建议

0 Success

<0 Used By Zeze

>0 User Defined. 自定义错误码时可以这样 (Module.Id << 16) | CodeInModule。

注意协议存储过程返回值使用同一个定义空间。

【不要捕捉异常】

如果你需要处理错误，捕捉处理完以后，再次抛出。

15) Zeze.Net.Service

网络的所有事件都通过这个类回调。

通过重载并override需要的方法进行特殊处理。

【线程】

网络事件直接在io-thread中回调，不要在回调中执行可能阻塞的操作。

如果需要请创建新的Task。

例子：

Zeze.Services.GlobalCacheManger,

Zeze.Services.ServiceManager,

Game2\linkd\gnet\LinkdService,Game2\linkd\gnet\ProviderService

Game2\server\Game\Server,

16) Zeze.Util.Task

这个辅助类提供执行并记录日志和统计的功能。

如果需要创建自己的Task，建议使用。

17) 一些建议

a) 定义模块级别的枚举(see Zezex/solutions.xml)。

一般用于模块处理构造全局唯一错误码。可以使用辅助函数Zeze.IModule.ReturnCode构造。

b) 客户端通讯尽可能使用Rpc。

框架在处理返回错误的时候自动发送Rpc的结果（rpc.SendResultCode(rc);）

异常的时候也会返回错误码（系统保留的负数的错误码）。

所以一般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return errorcode即可。基本没有需求需要catch。

18) 事务提交模式

a) CheckpointMode = Period

定时批量提交。可以缓存多次修改，一次提交。事务并发只依赖记录锁。

b) CheckpointMode = Immediately

每事务提交。事务执行完毕返回时，就意味着数据已经提交到后端存储数据库了。

重要应用可能需要这种模式。事务并发只依赖记录锁。

c) CheckpointMode = Table

可以在Table中启用选项 CheckpointWhenCommit="true"

当事务访问的记录启用上面的选项（根据所属的Table得到配置）以后，按 Immediately 提交，否则按 Period 提交。

事务并发：执行逻辑的时候依赖记录锁，Commit的时候需要锁记录所属的改名记录集合。并发性有一定降低。

这个模式比较灵活，适用面更广点。

d) CheckpointMode = PeriodNoFlushLock

定时批量提交。不需要FlushReadWriteLock。采用Table模式的实现方法，去掉全局FlushReadWriteLock。

它相当于配成Table模式，然后所有的Table都配置CheckpointWhenCommit="false"(默认）。

这个模式单独配置没有什么意义，写在这里仅仅为了说明。

\*)

配置

zeze.xml::CheckpointMode zeze.xml::CheckpointPeriod

性能考量

由于zeze的数据存储在后端数据库采用KV存储，一个记录只变了一部分，也需要整体写入，如果每次写入，存在一定浪费（流量，io）。

后端数据库支持的写入量。Period模式可以大大降低后端数据库需要支持的写入量。也使得浪费降低。

【注意】

选择不同模式应该主要根据需求来决定。

但是也要注意性能考量。

当在一个进程中同时运行多个Zeze.Application，并且运行同时访问多个App的事务时，

需要每个Application.Config.CheckpointMode 一致，这个目前没有很有效的检测手段，

所以是没有检查的。

# #### 更多说明

【不要捕捉异常】

如果你需要处理错误，捕捉处理完以后，再次抛出。

0) AutoKey：自增长key，仅支持 long 类型。

游戏经常需要分区，分成不同的服务器，然后又需要把人数降低以后的服务器合并。

如果对表格的key没有一定规划，合并的时候就很复杂。

提供一个自增长key，一开始就对规划范围内的服务器分配唯一的key，合并表格就不会冲突。

这在游戏分服运营，但是又需要合并时有用。

对于启用缓存同步，提供全球同服之类的系统，不需要使用这个。

1) 多数据库支持

提供多个 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。

配置参考：UnitTest\zeze.xml

2) 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，把属性 DatabaseOldMode 设为 1。

当需要时，Zeze 就会自动从老库中装载记录。

配置参考：UnitTest\zeze.xml

3) 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！）

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();

4) 缓存同步

多个 Zeze.Application 实例访问同一个后端数据库

一般的模式是后端数据库仅被一个 Zeze.Application 访问。

如果需要多个实例访问同一个数据库，需要开启缓存同步功能。

1) 启动 GlobalCacheManager

2) 配置 zeze.xml 的属性：GlobalCacheManagerHostNameOrAddress="127.0.0.1" GlobalCacheManagerPort="5555"

配置参考：UnitTest\zeze.xml

\*) 注意，不支持多个使用同一个 GlobalCacheManager 同步的Cache的 Zeze.Application 之间的事务。

参见上面的第3点。因为 Cache 同步需要同步记录的持有状态，如果此时 Application 使用同一个 Checkpoint，

记录同步就需要等待自己，会死锁。

\*) 由于逻辑服务器和GlobalCacheManager之间的连接非常重要，所以它们应该运行在一个可靠的网络中，

一般来说就是运行在一个机房中。

7)

9) 缓存同步 More

【Zeze 的性能来自缓存的命中率。】

当启用缓存同步以后，如果由于不同服务实例之间不停争抢数据，可能会造成命中率下降。

这时候需要做一定的规划。Zezex 提供了 ModuleRedirect 和 Transmit 支持用于提高命中率。

详细请阅读下面的文档或者代码。

a) Zezex\linkd.provider.txt

b) Zezex\README.md

c) Zezex\server\Game\Login\Onlines.cs

【不要捕捉异常】

如果你需要处理错误，捕捉处理完以后，再次抛出。

# #### 事务的划分

0. 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。

这里有两种派发选择，全部派发一个事务或者每一次派发一个事务，也可能派发在事务外执行。

例子：ChangeListener的使用。Zeze.Util.EventDispatcher

# #### 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。

有一次就问了 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。

# Zezex

使用探讨

## 【同一个世界同一个梦想】

不是很喜欢游戏分组的做法。

当然现在运营把不断开新服合并旧服，增加玩家游戏时间。我也没话说。

现在有了cache-sync，离这个梦想很近了。

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。

## linkd 转发方案说明

【名词】

linkd 转发服务器。

client 客户端，玩家。

server 服务器，主要逻辑服务提供者。现在作为linkd的provider。

module 模块，linkd根据module配置和转发。

provider 内部服务提供者，启动的时候注册相应的module到linkd。

【Provider】

每个provider是一个project，独立进程。

需要实现 solution.linkd.xml 里面的 Provider 模块中的协议。

每一个module只能在一个provider实现。

Provider 的 Service.type 一般都是 "server"，但是它主动连接 linkd，并注册支持的module。

\*\*\* Provider-Instance，运行进程实例，一般来说一个provider一个进程。

\*\*\* 启用cache-sync的server会运行多个进程。可以同时注册。

\*\*\* 某些特殊moudule可以存在于多个进程中，但不能同时注册，参见后面的动态绑定。

\*\*\* 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转发规则】

从收到的协议头中取出mouduleId，根据注册配置，选择一个provicer-instance。把协议转给她处理。

linkSid：client 在linkd上的连接编号，会在上下文中传给provider。

provider发送协议给client时，指定 linkSid。

【动态绑定】

由程序流程在需要的时候注册。

例子：

地图服务器实现了所有地图的管理，有多个进程实例。玩家一个时候只存在于一个地图实例。

enterworld：根据一定规则选择一个地图实例，该地图实例把自己注册到linkd的client.session中。

leaveworld：离开地图时，unbind。

【注意】 动态绑定是绑定某个玩家(client.session)的请求，不是全局的。

module默认是静态绑定的，动态绑定需要特别在配置文件"provider.module.binds.xml"中指明。

例子：

<module name="Game.Map" providers=""/> 不指定providers即可。

【总结一】

1) 每一个module只能在一个provider实现

2) 每个provider只有一个进程。

这样即可完成转发服务了。

【cache-sync】

启用 cache-sync 的 provider 可以运行多个进程，提供相同的服务。

此时，同一个module会注册多个provider-Instance。

linkd 可以随便选择provider-instance，目前轮转。

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。

同一个client的请求会被发给同一个provider-instance。

【provider.module.binds.xml】

【可选】没有使用动态绑定，也没有使用cache-sync，不需要这个配置文件。

参见文件内部的说明。

## 【性能】

0) cache-hit

TableCache 的命中率是决定性能的关键。

在cache-sync时，写会导致其他provider-instance的cache失效。

少量写，大多数读的模块数据仍然会有很高的命中率，不需要特殊处理。

大量写的模块数据需要提供特殊的解决方案，后面的hash模式，可以解决部分问题。

1) 并发性

游戏中，很多操作都是role相关的，具有局部性。是天然并发的。

虽然随意挑选provider-instance，但仍然会有很高的cache-hit。

对于公会成员列表这样的数据，多人共享，但不是全员共享，而且它的写操作很少，仍然具有很高的并发性。

2) 排行榜

【hash】

如果全局共享的数据可以按一定规则分组存储，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。

比如排行榜。table.key = account.hash % 1024。根据玩家账号hash分组，在自己的分组中排序，每一个分组数量都是1000，都要比需求大。

读取的时候汇总生成最终的排行榜，还需要提供自己的定时cache，只是定时更新。自己的cache和定时更新仍然很重要，否则读取操作也会导致cache降级。

这种模式的实现方案之一是提供特别的内部服务给server访问。每个服务都需要自己提供网络协议存储，有点工作量。

linkd实现了一种hash-choice模式用来简化实现，并且数据仍然存储在server的database中，具有cache-sync的优点。

开发的时候这种module的实现方式和普通的差不多，提供技术手段实现请求转发。

【要点】

1) 当然排行版的数据修改一般在新的事务中执行。特殊情况下，并且少量的话，可以嵌套到原来事务中。

2) 修改排行榜需要的数据都可以通过参数传递，不需要自己去其他数据表里面读。

3) 没有返回值，异步的。不需要等待排行榜数据的更新。特殊情况下，可以等待。

【hash分组数量】

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1024。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（作废掉重来）。

分组数量是比较关键的，要慎重考虑。

【TaskOneByOneByKey】

对于每一个分组，是不需要并发的。可以使用TaskOneByOneByKey.Exceute(hash, task)。

在这里可以再次设置并发执行的"度"，TaskOneByOneByKey默认构造了1024的并发度。

这是运行期配置，修改不影响数据。

【Task.Run】

这里任务实际提交给系统线程池。可以配置系统线程池的线程数量。zeze.xml。

这是运行期配置，修改不影响数据。

【最大并行机器数量】

= hash分组数量 / cpu内核线程数

假设所有的cpu线程都满载。

这是运行期配置，修改不影响数据。

【实现】

当其他module调用排行榜时，

1) 排行榜接口先判断是否要嵌套到原事务中，如果要就本地执行（此时cache-sync起作用）。这种调用不能太多。

2) 判断调用者和自己是否同一个服务(same AutoKeyLocalId)，决定是否转发请求给相应的服务器或者本地NewProcedure。

Linkd ChoiceHash

当可用服务器数量发生变动时，负载分配会重新洗牌，Cache命中率会下降。由于一致性缓存，事务执行仍然是正确的。经过一段时间，Cache命中率就能提升。当然如果可用服务器变动很快，就会有问题。但谁会没事，随时随地增加减少服务器呢。

游戏

. 建议命名规范

Bean用'B'开头。

客户端发送服务器处理的协议用'C'开头。

服务器发送客户端处理的协议用'S'开头。

Table用't'开头, 并且小写。

## ModuleRedirect

这个主要目的是为了提高共享模块的 cache 命中率问题。see linkd.provider.txt 排行榜相关问题。

【例子和说明】

[ModuleRedirect()]

public virtual TaskCompletionSource<int> RunSomeMethod(int param, Game.TransactionModes mode = Game.TransactionModes.ExecuteInAnotherThread)

{

int hash = Game.ModuleRedirect.GetChoiceHashCode();

return App.Zeze.Run(() => SomeMethod(hash, param), nameof(SomeMethod), mode, hash);

}

1) [ModuleRedirect()]: 注解声明，表明需要转发支持，去掉就不会被转发。

2) virtual: 必须的，否则生成的时候会抛异常报错。

3) mode: 这个参数可选，不提供的话，ModuleRedirect 默认按 ExecuteInAnotherThread 处理。不支持ExecuteInAnotherThread的实现是没必要声明ModuleRedirect的。

4) return: TaskCompletionSource<int>，调用者可以等待调用完成； 或者void。【推荐 void，不关心处理结果】。等待其他存储过程结束是很危险的，可能会导致死锁。

5) RunSomeMethod： 方法名必须以"Run"开头，否则生成代码时会报错。这是为了区分普通模块接口和可能起一个新事务的接口。

\*) 接口方法的实现功能一般为：计算hash；传递一下参数给真正的实现；决定使用什么模式执行存储过程；决定是否使用TaskOneByOne（App.Zeze.Run的最后一个参数）。

具体的实现逻辑在后面这个方法中。

protected int SomeMethod(int hash, int param)

{

// 真正的实现代码

return Procedure.Success;

}

1) SomeMethod: 开放接口名字去掉"Run"，剩下的部分。

2) return: 返回存储过程处理结果，跟协议处理 ProcessXXX 一样的定义。

3) params: 【第一个是数据分组hash】；其他是自定义参数，和真正开放方法的参数一致，但不包括最后的mode。

4) 注意: 这个函数被调用时可能，上下文中可能没有 Login.Session。如果实现需要这个了再说。

5) protected: 子类能调用，建议不开放。

【总结】

规范要求实现分成两个方法，这是为了实现简单，也保持灵活，还有效率。

ModuleRedirect 通过生成模块的子类来实现转发请求。

如果仅通过调用 base.RunSomeMethod 来实现，mode就不能省掉。因为底层需要根据情况修改mode的值。

【接口方法返回数据的两种方案】

1) 同步模式，通过 out ref 参数返回数据。内部需要等待实际存储过程执行完毕。有死锁风险。

2) 异步模式，通过增加一个回调，"Action<int> resultCallback"。内部需要返回数据时，回调这个接口。

这个模式使用时，要注意，resultCallback一般在另一个事务中回调。注意表中的数据跨事务传递的问题。

. ModuleRedirectWithHash

根据指定的 hash 转发请求。第一个参数必须是int hash。

内部转发的时候使用指定的 hash，而不是根据Session计算。

除此外，其他和 ModuleRedirect 一样。

. ModuleRedirectAll

遍历处理所有的 hash 分组。执行的效果和MapReduce类似。这里更加专用化。

由于每个 hash 分组都可能有返回值，所以不能使用ref|out返回数据，只能使用callback。每个分组分别回调。

a) ModuleRedirectAll 的接口方法参数如：(..., Action<...> onHashResult, Action onHashEnd)

onHashResult 如 Action<long, int, int, ...> 用来处理hash分组的结果。hash分组的处理没有返回值时，不需要这个参数。

1) 第一个模板参数是long sessionId，用来区分不同的调用。

2) 第二个模板参数是int hash，用来区分不同的hash分组。

3) 第三个模板参数是int returncode，hash分组的处理结果，只有Success时，自定义参数才有效。

4) ... 自定义参数。

onHashEnd 类型必须是 Action<ModuleRedirectAllContext>，当所有的hash分组都处理完的时候回调，不关心处理完成情况时，可以不定义这个参数。

b) ModuleRedirectAll 的实现方法参数如：(long sessionId, int hash, ..., Action<...> onHashResult)

实现方法不需要 onHashEnd 参数。

sample: see Game/ModuleRank/RunGetRank

【ModuleRedirect 汇总方案选择】

当需要遍历所有的hash分组时，可以使用 ModuleRedirectAll，也可以直接从数据库中读取。

采用 ModuleRedirectAll 时，可以把hash分组的读也分配到相关的配置服务器中，不会破坏缓存，具有很高的命中率。

具体采用哪种方案，需要根据具体需求来决定。一般建议如下：

1) 当分组数据量不大的时候直接从数据库中装载，不使用ModureRedirect。see server\Game\ModuleRank.GetRank.

2）当分组数据量比较大，但是处理结果的数据量比较小，此时ModuleRedirectAll比较适合。

3) 原则上所有的读取结果都可以在本地缓存，定时更新，此时上面两种方案都比较高效。如果需求不能使用定时缓存，那么ModuleRedirectAll更合适。

ModuleX

{

//////////////////////////////////////////////////////////////////////////////////

// 全局单一模块分组增加并发度

// 某公司交易量非常大，单个共享的账号（就是一个long）并发度不够了。

// 那么可以分成多个子账号。需要总金额时，遍历汇总。

Table<int, Money> subMoneyAccounts; // key = hash%concurrentLevel

int concurrentLevel = 200; // 分200组。

//////////////////////////////////////////////////////////////////////////////////

@Redirect()

public void RunAddMoney(int x, RedirectResultHandle<EmptyBean> resultCallback) {

// 接口实现：默认直接调用实际实现。

// 默认按当前Login.Session的account.hashCode。

// 这样限制了这个方法只能在有Login.Session上下文的情况下使用。

// 如果没有Login.Session，或者需要指定特别的hashCode，可以在上面的注解里加参数。

// 如：@Redirect(ChoiceHashCodeSource="GetSetXHashCode()")

// 其中，"GetSetXHashCode()"是代码会被生成到ModuleX的子类中，

// 所以这个代码在子类的重载函数中需要能正确编译并且执行。

// 一般是ModuleX的protected级别以上的实现即可。

// 注意，如果GetSetXHashCode需要参数，这个参数需要在RunSetX(Object hashCodeContext, int x)中声名，

// 并且由于java丢失了参数名字，只能根据它的位置叫，如，arg1。

int hash = Zezex.ModuleRedirect.GetChoiceHashCode();

// App.Zeze.Run 包装了几种执行方式。不使用的话，可以直接调用SetX();

// 这样如果请求没有没有重定向，那么将会嵌入到调用者事务内执行。

// 如果发生了重定向，转到另外的服务器执行，就是独立事务了。

// 这样会有点不一致发生。为了语义更加明确，建议用ExecuteInAnotherThread。

// 除非有特别需求。

App.Zeze.Run(

() -> AddMoney(x, resultCallback),

"RunSetX",

Zeze.TransactionModes.ExecuteInAnotherThread,

hash);

}

protected void AddMoney(int hash, int x, RedirectResultHandle<EmptyBean> resultCallback) {

var key = hash % concurrentLevel;

subMoneyAccounts.getOrAdd(key).Money += x;

resultCallback.run(new EmptyBean()); // result callback 会最终回调RunAddMoney调用者的resultCallback。

}

//////////////////////////////////////////////////////////////////////////////////

@RedirectWithHash()

public void RunAddMoneyWithHash(int hash, int x, RedirectResultHandle<EmptyBean> resultCallback) {

// hash 由调用者指定hash，所以没有Login.Session的问题，就是对调用者来说，有点负担。

// 可以作为内部实现，在外面包装一个public接口，把hash的获取解决了。

App.Zeze.Run(

() -> AddMoneyWithHash(hash, x),

"SetXWithHash",

Zeze.TransactionModes.ExecuteInAnotherThread,

hash);

}

protected void AddMoneyWithHash(int hash, int x, RedirectResultHandle<EmptyBean> resultCallback) {

AddMoney(hash, x, resultCallback);

}

//////////////////////////////////////////////////////////////////////////////////

// 把请求重定向到指定的服务器

@RedirectToServer()

public void RunRedirectToServerSample(int serverId, RedirectResultHandle<EmptyBean> resultCallback) {

// serverId 调用者根据需求决定。

App.Zeze.Run(

() -> RedirectToServerSample(serverId, resultCallback),

"RunRedirectToServerSample",

TransactionModes.ExecuteInAnotherThread,

serverId);

}

protected final long RedirectToServerSample(int serverId, RedirectResultHandle<EmptyBean> resultCallback) {

resultCallback.run(new EmptyBean()); // result callback，会最终RunRedirectToServerSample的调用者的callback。

return 0L;

}

//////////////////////////////////////////////////////////////////////////////////

// 总结

// 1 每种重定向都有两个函数组成。一个通常public公开的，一个通常protected实现。

// 2 @Redirect @RedirectWithHash 基本目的一样，只是获取hash的方式不同。

// 由于@Redirect默认依赖Login.Session，所以建议直接使用@RedirectWithHash完全自己控制hash。

// 需要给调用者省去计算hash的麻烦，就另外写一个自己的接口函数。

// 3 重定向方法返回值是可选的，不需要返回结果，不定义RedirectResultHandle参数。上面的是例子。

// 由于java模板丢失了类型，所以RedirectResultHandle内的Bean类型需要注册。

// 比如在Module.Start() { super.putClassForMethod("RunRedirectToServerSample", EmptyBean.class); }

// 4 @RedirectAll(GetConcurrentLevelSource="GetSubAccountConcurrentLevel()")

// 这个给所有的hash分组广播请求，并行处理，然后返回结果，最后汇总。

// 一般在数据集比较大结果集比较小的情况下用。这个东西类似google的MapReduce.

public int GetSubAccountConcurrentLevel() {

return conrurrentLevel;

}

}

. client 选择 unity+ts

尽量采用 rpc？

. map 动态绑定

test git tag and checkout

# Git

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

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