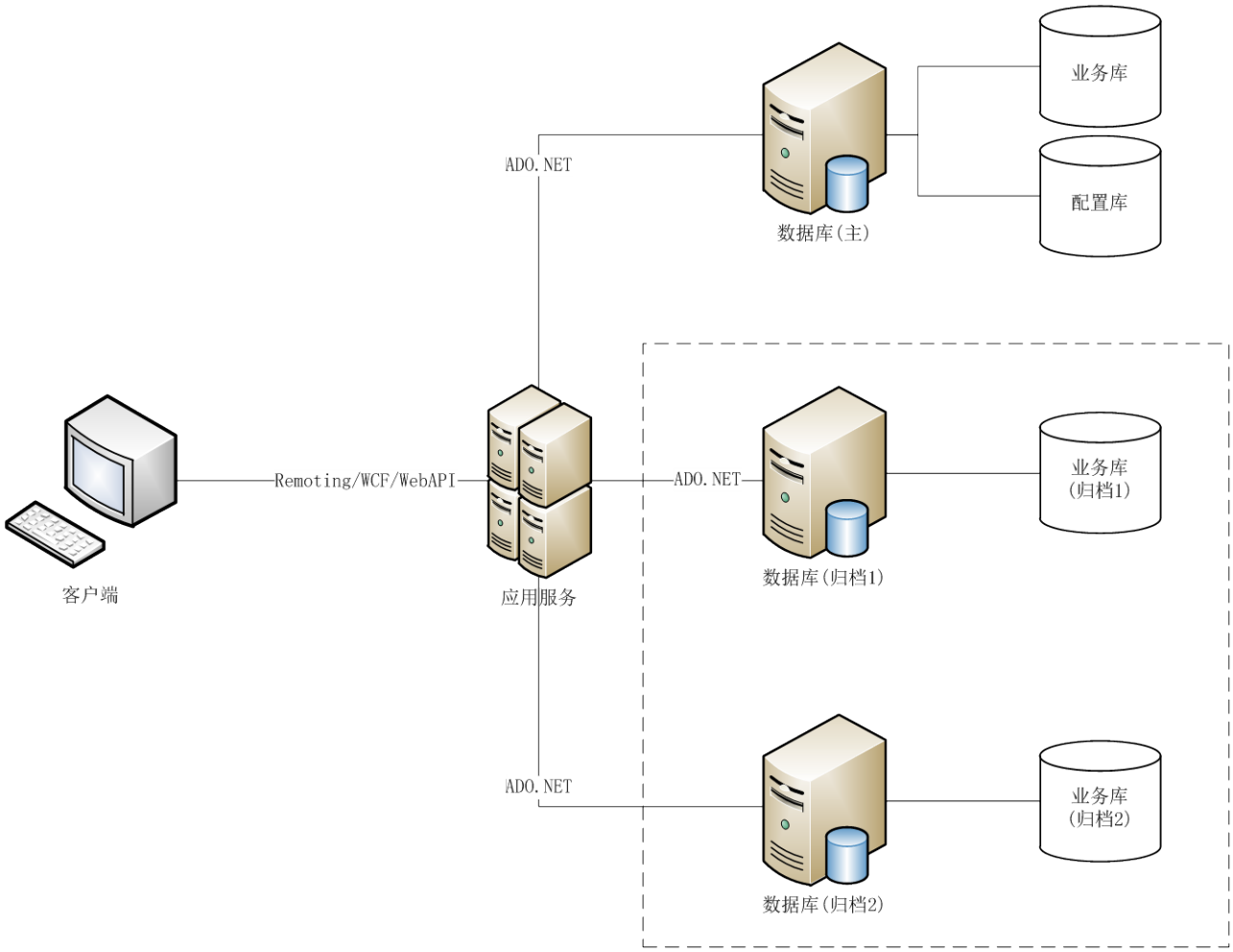
24 分库归档及查询

在实际业务场景下，业务数据（比如订单）始终在累积、不断增长（比如超过 1Td 的历史数据），到了一定量后必定会影响到系统的性能，进而影响到用户的体验，以至于不可接受。我们在系统设计时就应考虑到这方面的应对策略，制定相应的（不活动的）历史数据归档方案。

历史数据归档手段有很多，本章节专注于探讨可透明化归档和查询历史数据，突破单机数据库的存储空间、CPU 和 I/O 性能等不再能继续（垂直）扩展带来的瓶颈问题（比如当数据库表分区等技术都已承受不住压力的时候），将主库中不再需要被更新、待归档的大表记录按照一定的分片规则，均衡地迁移至一组历史数据库里。这样，即能保证主库的整体性能，也能方便从历史库中查询到归档记录，合并为一个数据集供浏览。

24.1 架构实现

Phenix 分库归档及查询技术，可实现的架构如下所示：



实际生产环境下，同时还应做好主库和历史库的容灾策略。虚线内的服务器不一定是 2 台，但 2 台是历史库的最低部署要求。

24.2 前提条件

为透明化归档和查询历史数据，开发者需要做出一定的配合来解决以下问题。

24.2.1领域模型设计问题

领域驱动设计是 Phenix丶提倡的、实现复杂业务逻辑的一种非常好的方式，我们应该利用 DDD 聚合的概念拆分领域模型。好的领域模型，有利于 Phenix丶历史数据归档方案的应用，有利于规避并发、分布式事务等这些让开发复杂化的技术应用。

领域模型由一个或多个 DDD 聚合组成，我们只要合理地划分和设计聚合，就不会产生任何并发和事务性问题，没有必要使用分布式事务。原子级的处理数据的变化不应该溢出到聚合之外，也就是说数据的事务处理应该在一个聚合内完成，聚合是作为原子级数据处理的一个单元（此处衍生出一个“如何通过事件来维护聚合（和服务）之间的数据一致性”的问题，将在后续章节中专题讨论）。

聚合是实体对象、值对象及其聚合根的集合，每个聚合都有且仅有一个聚合根，我们务必要通过聚合根来访问领域对象进行数据的处理，而不是随意操作里面某个普通的对象来做持久化。

聚合根可以是一个 Root 对象，也可以是一个 RootList 集合对象，如下图（在“业务结构对象模型”章节中也有展示）：

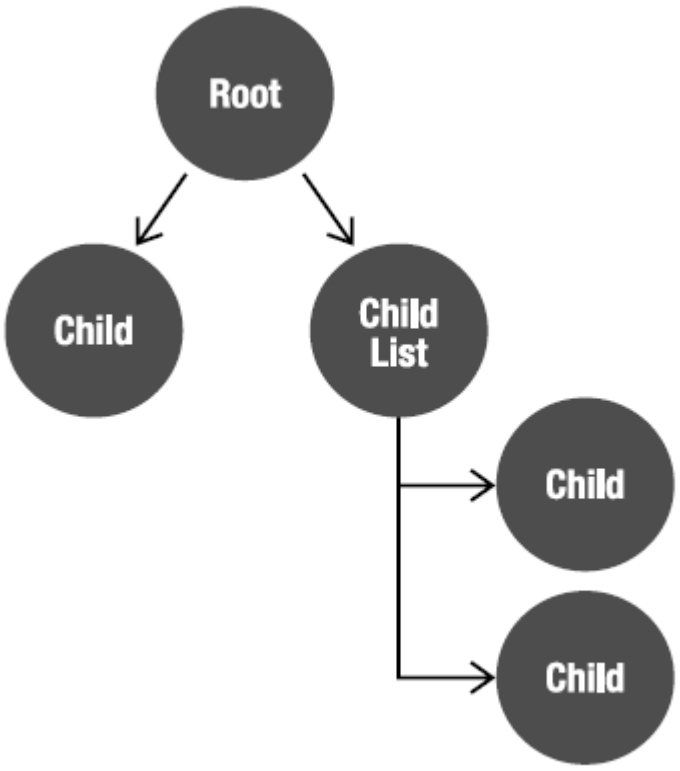


Figure 4-2. *Object graph illustrating containment*

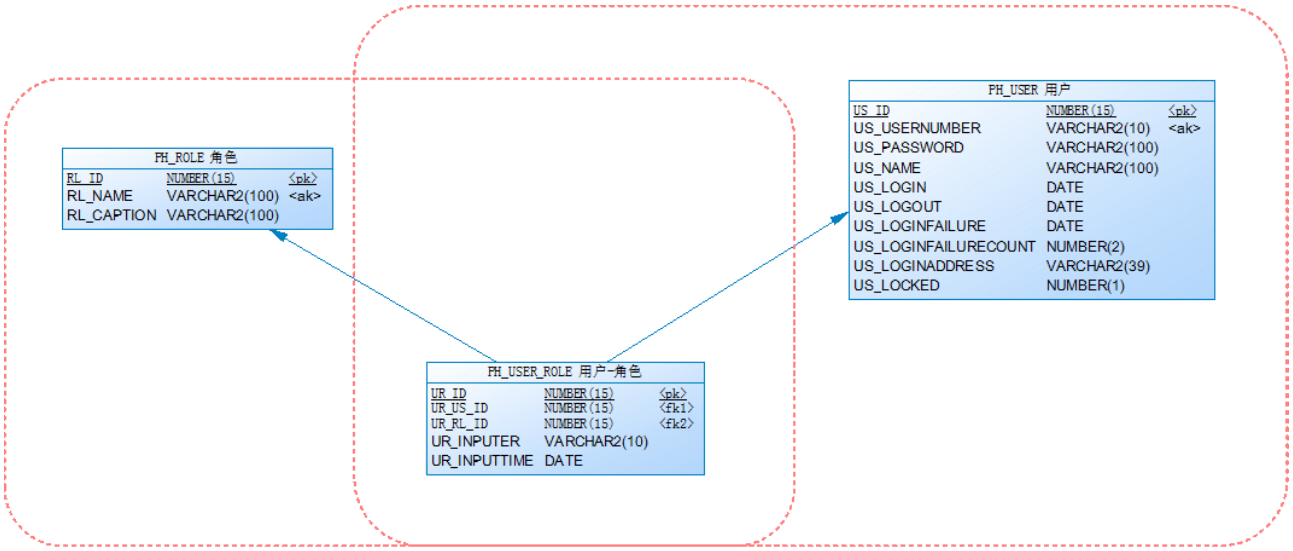
历史数据的归档，Phenix丶是通过访问聚合根来实现的，然后逐层遍历并归档有聚合关系的从业务对象（仅 CompositionDetail，忽略 AggregationDetail）。

举例如下，会归档 User 和 UserRoleInfo 对象的数据：

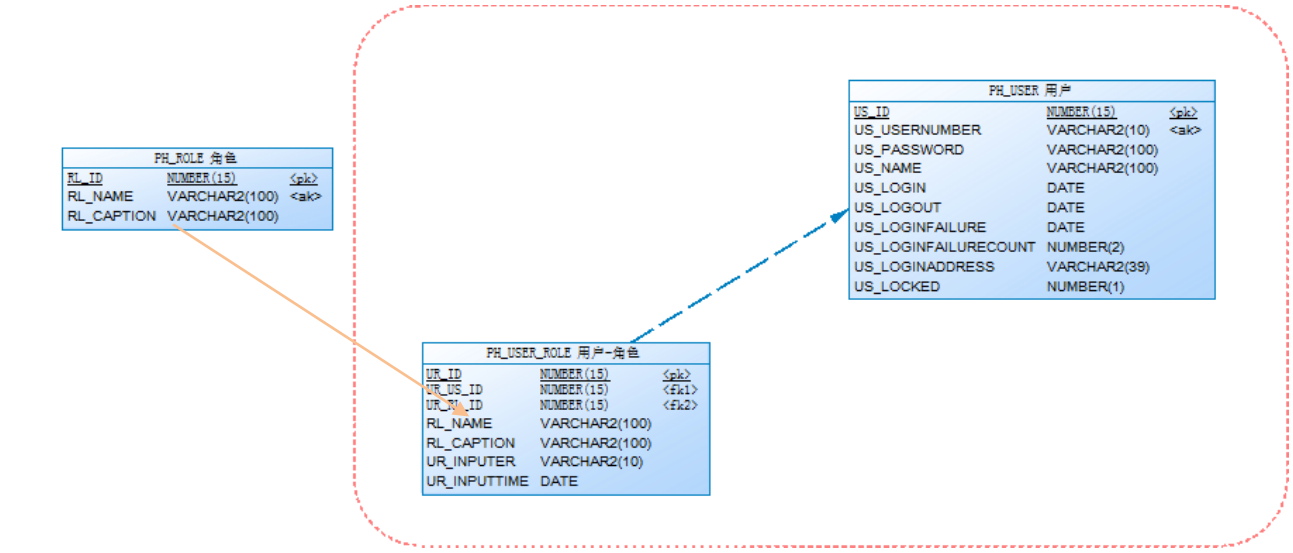
```
/// <summary>
/// User
/// </summary>
[Serializable]
public class User : User<User>
{
  private User()
  {
    //禁止添加代码
  }

  #region 属性

  //* 组合关系的从业务对象集合
  /// <summary>
  /// 用户角色
  /// </summary>
  public UserRoleInfoList UserRoleInfos
  {
    get { return GetCompositionDetail<UserRoleInfoList, UserRoleInfo>(); }
  }

  #endregion
}
```

24.2.2数据模型设计问题

一般情况下，主库的数据模型设计是要迎合数据库及其产品的客观条件的。比如遵循关系型数据库设计的三大范式，目的是减少冗余、保证结构的合理，而且还要考虑到设计是否有利于数据库产品的存取及其效率的优化。但是，领域模型迎合的是业务逻辑，体现的是业务类之间的关系。比如，同一个数据在某个聚合内它可能被映射为一个实体对象（进而可能是聚合根），而在另一个聚合内它可能被映射为一个值对象，这是由不同的实际业务场景决定的。所以，领域模型很难做到与数据模型的一对一地映射，需要依靠 Phenix丶对象关系映射引擎来管理它们复杂的映射关系，自动化完成数据的存取（请参考“Addin 工具使用方法”、“业务结构对象模型”等章节）。



在历史数据的处理上，Phenix 建议将数据模型的设计尽可能地迎合领域模型，支持更深层次的数据集成、数据共享（比如数据仓库）方面的需要。这样，虽会带来一定程度的历史库数据的冗余，但却可以方便、高效地在历史库里进行多维数据的处理、抽取。



以上图示，假设是要归档‘用户’业务树，不归档‘角色’业务树的话，可以把‘角色’表里必要的数据冗余到历史库的‘用户-角色’表中。但是，如果是对现有已稳定的系统进行归档改造的话，建议主库和历史库的表结构尽可能保持一致（历史库可以打掉物理主外键关系），可以最大程度地复用现有系统的查询统计功能。

以上两种方案，不管主库和历史库的表结构保不保持一致，它们的表名和字段名都必须保持一致。如果有冗余，冗余字段的命名必须和被冗余表的字段名保持一致（参考以上图示）。

24.2.3数据模型映射问题

默认下，Phenix 假设主库和历史库的表结构是完全一致的，除非你在关系映射中显式申明有哪些不一致的地方，用 FieldAttribute 标签的相关属性进行说明：

属性	说明	备注
IsRedundanceColumn	指示该字段是冗余字段	缺省为 false； 如设置为 true： ■ 归档数据时，Phenix丶根据 FieldAttribute.TableName 和 FieldAttribute.ColumnName 申明的主库表字段取值，保存到 ClassAttribute.TableName 和FieldAttribute.ColumnName 申明的历史库表字段里； ■ 查询历史库时，Phenix丶根据 ClassAttribute.TableName 和 FieldAttribute.ColumnName 申明的历史库表字段取值；

24.2.4系统持续升级问题

为保证系统在整个生命周期内是可以被持续升级的，数据库应该向后兼容，考虑向前兼容。

在开发者对系统做出变更时：

- 建议要能做到先升级历史库，再升级主库，升级程序，最后进入新的系统稳定态，三步动作无需连贯执行。期间，只在短时间内中断被升级部件的运行，其他部件不应受到影响。
- 与业务有关的，数据库表结构只能新增字段，不能删除字段，也不能修改已有字段的定义、更换字段值的含义（比如枚举 Flags 只能递增不能‘夹塞’），并且新增字段必须有默认值。
- 历史库的数据结构，在系统稳定态下应该与主库保持一致，且允许添加一些冗余结构，以便直接映射领域模型。
- 同一套历史库组合，在系统稳定态下应该保证每个数据库的数据结构都是完全一致的。
- 历史库的数据结构，应去掉无用的约束条件（如主外键关系）、触发器，以便在归档多个有关联的聚合数据时，可以不用考虑先后次序。
- 对于必须修改原有数据库结构，不得不全部或一定范围内的停机，则交给 DBA 操作，不纳入持续交付流程。

24.2.5规范化约定

Phenix丶在第一次连接主库时，会按照以下约定（且数据库类型、登录口令都要一致）顺着递增的序号尝试连接历史库，直到连接不上下一个序号为止，最终能被识别出多少就多少个，除非重启程序（三层架构下的 Host）才会被重新扫描一遍。

业务系统在运行期间，应保证主库、历史库都处于正常运转状态，归档时如果连不上是会被抛出异常的。

24.2.5.1 历史库数据源命名的约定

历史库的数据源名称（指 Data Source，Oracle 为数据库连接串，SqlSever 为数据库服务器名）等于主库的数据源名称加后缀从 1 起递增的序号。比如，主库的连接串为 TPT，那如果历史库为 2 台一组的话，则分别为 TPT1、TPT2：

```
TPT =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.248.50)(PORT = 1521))
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = TEST)
    )
  )

TPT1 =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.248.51)(PORT = 1521))
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = TEST)
    )
  )

TPT2 =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.248.52)(PORT = 1521))
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = TEST)
    )
  )
```

24.2.5.2 历史库登录用户名/数据库名的约定

历史库的登录用户名（指 Oracle 的 User Id）/数据库名（指 SqlSever 的 Initial Catalog）等于主库的加后缀从 1 起递增的序号。比如，Oracle 主库的 User Id 为 admin，那如果历史库为 2 台一组的话，则分别为 admin1、admin2。

24.2.6注册历史库

如果在现有系统环境下无法做到以上规范化约定（但起码数据库类型、登录口令都要一致），也可以编写一个插件程序集添加到 Host 里来实现一样的效果。方法是在插件类的初始化函数里调用

Phenix.Core.Data.DefaultDatabase 类的 DbConnectionInfo 属性的 AddHistory()函数：

```
/// <summary>
/// 添加历史库连接串
/// </summary>
/// <param name="dataSource">数据源</param>
/// <param name="userId">用户ID</param>
/// <param name="initialCatalog">数据库名(MSSQL)</param>
public int AddHistory(string dataSource, string userId, string initialCatalog)
```

比如插件类可以这样写：

```
public class Plugin : PluginBase<Plugin>
{
  /// <summary>
  /// 初始化
  /// 由 PluginHost 调用
  /// </summary>
  protected override IList<MessageNotifyEventArgs> Initialization()
  {
    Phenix.Core.Data.DefaultDatabase.DbConnectionInfo.AddHistory("TPT1", "admin1");
    Phenix.Core.Data.DefaultDatabase.DbConnectionInfo.AddHistory("TPT2", "admin2");
    return new List<MessageNotifyEventArgs>() { new
MessageNotifyEventArgs(MessageNotifyType.Information, "注册历史库", String.Format("注册数量 = {0}",
Phenix.Core.Data.DefaultDatabase.DbConnectionInfo.HistoryCount)) };
  }
}
```

插件的开发和部署方法，参见工程里的《Phenix Framework Tools.权限管理.11.自动解除用户登录失败锁工程.服务插件》文档。

24.2.7指定分库拆分键

Phenix丶分库策略采取的是横向分表，利用到了切分表数据的表过滤器技术。

Phenix丶提供的表过滤器管理工具，见 Phenix.Security.Windows.TableFilterManage 工程，用于指定可匹配的表字段作为归档的拆分键（SplitKey）：

只要业务树 Root 对象里有匹配的字段映射关系（FieldAttribute 的 TableName+ColumnName、LinkTableName+LinkColumnName 与 TableFilter 配置的表字段一致），Phenix丶就会提取这些匹配字段的值进行 Hash 取模，确定将这棵业务树归档到哪个历史库，或（如果有多个匹配字段的话）冗余到哪几个历史库。

以下是与归档拆分键有关的 FieldAttribute 标签属性：

属性	说明	备注
TableName	指示该字段对应的表名	如果为空，则认为从类的 ClassAttribute.TableName 获取；
ColumnName	指示该字段对应的表列名	如果为空，则认为从字段名获取，获取方法为：字段名第一个字符如果为'_'将被裁剪掉；
InTableFilter	用于表过滤器	缺省为 true；

以下是与归档拆分键有关的 FieldLinkAttribute 标签属性：

属性	说明	备注
TableName	关联表的表名	
ColumnName	关联表的表列名	

IsValid	是否有效	缺省为 true；

如果未使用表过滤器技术或没有匹配到，Phenix丶会利用 Root 对象的主键（FieldAttribute 的 IsPrimaryKey = true）作为归档拆分键：

属性	说明	备注
IsPrimaryKey	指示该字段是主键	缺省为 false；

所以如果你对数据是否归档到哪个历史库没有特别要求的话，表过滤器技术并不一定要用到。

24.2.8谨慎设计切分模型

Phenix丶支持横向分表分库策略下的切分模型，自动实现的是跨库的横向合并（union）查询，但并未实现跨库的纵向合并（jion）查询（需开发者自行实现）。所以，你在设计切分模型时，不光要考虑到能否将数据均衡地迁移到每个历史库中，也要尽可能避免不必要地将业务树冗余到多个历史库中，要将未来可能的联合查询限制在一个库内。因为一旦推上线的话，你的系统多多少少都要被锁定在这个切分模型中，之后再要调整方案、重新切分就并非易事了。

24.3 归档方法

请直接操作 Root 业务（或集合）对象的归档函数：

```
/// <summary>
/// 归档
/// </summary>
public void Archive()

/// <summary>
/// 归档(运行在持久层的程序域里)
/// </summary>
/// <param name="connection">主库连接</param>
public void Archive(DbConnection connection)
```

在执行以上语句之前，先要 Fetch 出 Root 业务（或集合）对象供调用（无需遍历它所有层级的 Composition 从业务对象到内存中，因为 Phenix丶在执行过程中会逐层自动遍历它们并归档，只要在业务类里有相关的属性即可）。

从业务对象的属性申明，类似如下写法：

```
/// <summary>
/// User
/// </summary>
[Serializable]
public class User : User<User>
{
  private User()
  {
    //禁止添加代码
  }

  #region 属性

  //* 组合关系的从业务对象集合
  /// <summary>
  /// 用户角色
  /// </summary>
  public UserRoleInfoList UserRoleInfos
  {
    get { return GetCompositionDetail<UserRoleInfoList, UserRoleInfo>(); }
  }

  #endregion
}
```

示例：

```
//获取
adminUser = User.Fetch(p => p.Usernumber == Phenix.Core.Security.UserIdentity.AdminUserNumber);
//归档
adminUser. Archive();
//标记删除
adminUser.Delete();
//提交删除
adminUser.Save();
```

24.4 查询方法

业务（或集合）对象的 Fetch()方法，如果没有在类上申明 Phenix.Core.Mapping.HistoryAttribute 标签的话，就是构建自主库。

Phenix.Core.Mapping.HistoryAttribute 标签的属性：

属性	说明	备注

FetchAll	构建自全部库	缺省为 false； true：主库+历史库 false：历史库

示例代码如下：

```
/// <summary>
/// User（历史库）清单
/// </summary>
[History(FetchAll = false)]
[Serializable]
public class UserHistoryList : Phenix.Business.BusinessListBase<UserHistoryList, User>
{
}

/// <summary>
/// User（主库+历史库）清单
/// </summary>
[History(FetchAll = true)]
[Serializable]
public class UserHistoryAllList : Phenix.Business.BusinessListBase<UserHistoryAllList, User>
{
}
```

调用以上这两个类的 Fetch()方法，可分别构建出历史库的业务集合对象、主库+历史库的业务集合对象。