**数据服务器的分析与设计**

**一、需求分析**

数据服务器是一个数据存取服务，它的作用是把客户需要的所有数据存储于本系统，使客户端不关心数据的持久化、备份，并提供高速的数据访问服务。数据服务器对数据的存取是与业务无关的，也就是说在数据存取的过程中，数据服务器不关心数据的内容和意义，完全由客户自行存取、自行解析、自行处理。

**(一) 项目功能分析**

对一般日弱交互游戏及我们项目的数据部分进行分析后，可以把数据分为以下几类：

1. **用户基础数据**

用户基础数据是指用户属性与基础功能数据，我们可以把它分成：

<1> 静态数据。例如用户的guid、性别等等。

<2> 动态数据。例如用户的生命值、等级等等。

<3> 会话数据。例如用户的sessionkey等等。

<4> 多组高级数据。例如用户的技能数据、装备数据等等。

由于随着系统功能的复杂，用户数据会越来越大，所以我们把用户数据分成小块，让这些数据实现快速存取。

1. **用户战斗数据**

<1> 用户战斗列表。现有项目中的战斗是1对1的，也就是用户可以和任意的其它人开启多场战斗，这样用户就应该有一个属于自己的战斗列表。

<2> 战斗数据。每场战斗是属于两个人的，所以这是一个公共数据，需要单独存储。现有项目中的战斗是回合式战棋的战斗，每个回合可以进行五步操作，而战斗开始时应该有两个用户的初始数据。这样战斗数据就要分为：战斗初始数据和战斗回合数据表（应该是一个队列表的数据结构）。

1. **用户在线匹配列表**

用户在线匹配列表是用户快速在线战斗的功能，需要用户进入这个功能后，自动匹配用户进行游戏，这种匹配列表可能分为不同的等级或战斗力评分的列表，例如每十级一个列表等等。

这个数据是一个公共数据，主要是队列的插入与取出功能。

1. **各种排行榜**

排行榜是用户各种能力的排序，在排序数据变更时，需要去更新排序键值以更新排行，要求实现一个键值排序表。

小结下，系统需要把大数据分成小块进行存储，系统需要支持列表、排序表的数据结构。而我们的数据量一定不是单台服务器可以存储的，这就要求我们把存储做成分布式系统，而如果我们把所有数据存储到数据库中，由于需要频繁对数据进行存取，这会成为数据访问的瓶颈所在，所以我们对数据需要做缓存，又由于数据量的问题，缓存也必须是分布式的。分布式系统中，key-value数据格式的分布式效果最好，而key-value也正好比较适合把大数据拆分成小块（这样分块存取可以有效降低带宽占用）进行分布式存储。所以，我们的数据服务器的数据就要是基于key-value格式的，由于还有列表和排序表的需求，我们的数据就要支持列表和排序。

**(二)缓存与持久化分析**

1. **缓存部分**

我们系统支持的数据主要为key-value格式的分布式数据，memcached和redis是目前最优秀的两个开源缓存服务，他们在缓存系统中的表现是不相上下的。我们的数据中需要最好有原生的支持列表和排序表的数据结构，而memcached只支持最基本的key-value数据格式，而redis原生支持了列表、排序表、哈希表等数据结构，所以redis是最符合我们的需求的。

对于缓存而言，它是运行在系统中的服务器程序，所有数据均在内在中，一旦遇到宕机、断电，数据会全部丢失，这就要求我们的缓存在重启后能自动恢复，而自动恢复的方法就是由客户端访问数据时，触发对持久化的访问，然后把持久化的数据缓存到内存中，也就是说不主动恢复，由客户端的访问触发被动恢复，这样做比较简单，而且并对缓存的命中率较高。

1. **持久化部分**

对于key-value系统（统称为nosql数据库），市面上有一些如mongodb、memdb等合适key-value数据结构的数据库产品，而对于我们熟悉的mysql，它并不是原生支持key-value数据格式，从这个角度来看，nosql类数据库明显更好一些，但对于我们现有的人员和技术，对nosql数据库的了解少之又少，对其性能、安全性以及周边工具都不了解，所以用mysql去实现key-value结构是一个相对划算的方向。

对于mysql，我们需要做的就是怎样让它像缓存一样支持key-value、列表、排序表的数据结构。

key-value对于关系数据库而言就是一张二维表，这个二维表有两个字段，一个是key（字符串类型），另一个是value（二进制类型），由于缓存支持的key-value是把数据散列到不同的缓存节点的，每个节点中都是用哈希表或红黑树之类的能快速地根据key定位到value的数据结构来提高访问效率的，而对于mysql这种不是原生支持key-value的数据库，只能哈希到节点，节点内的定位就只能是顺序查找了，效率非常低，因此，我们需要解决的问题是单个数据库的数据访问效率。其实我们可以把每个数据库中预先建立多张key-value表（如999张表），再使用哈希定位到某个数据库中后，再用取余的方法哈希到这999张表的某一张，当然于我们固定地把数据分开，这样查找效率就提高了999倍，如果我们平均每个数据库的数据量为10万条，那么我们的最差查找次数也只有100次而以，再加上mysql本身的索引功能，我们使用mysql对于节点的数据访问效率是不成问题的。

对于基于key-value格式的列表，我们使用动态建立的方式，也就是说，需要列表时，我们动态建立一张二维表，表名就是以key作为名称，这样可以很好的散列，value就是这个二维表本身，这个表只有一个字段，是一个二进制类型，有插入的需求时，就向这个表中插入一行，就形成了一个列表，对于插入、删除，都是非常方便的。

对于基于key-value格式的排序表，我们用和列表类似的方式，不同的是这个张二维表有两个字段，一个是排序键值，另一个是二进制类型的元素，这样每插入一个元素，就是一行，这个表按排序键值进行排序就可以得到一个有序表了。

还有一种数字类型，主要用来做ID分配，这类数据可以在每个数据库节点中建立一张专门的二维表，一列为key，别一列为当前数字，需要ID分配时，查找到相应的key后，把这个数字加一返回即可。

对于列表和排序表两种需要动态建表的，由于有动态增加数据库结点的需示，所以要建立key的路由服务器（用于确定key所在的节点），这就要求我们在系统重启时能确定动态建立的列表和排序表的key分布情况，所以在每个数据库节点上建立一张专门用于记录动态建立的二维表的信息，我们可以建立一张二维表，第一列是key（也就是动态表的表名），第二列就是表类型（列表或排序表，将来可能会支持更多的类型）。

**(三)小节**

经过以上的分析，我们确定了：

1. 使用以key-value为基础的数据格式，使用redis作为缓存，使用mysql作为持久化。

2. 数据写入时持久化同步缓存，数据读取时先缓存读取、再持久化读取（并同步到缓存）的方式。

3. 缓存和持久化都支持动态增加节点。

4. 数据结构支持key-value的基本数据、数字分配、列表、有序表，这些数据要支持存储方式（只缓存、只持久化、缓存加持久化）。

**二、系统设计**

整个数据服务器应该包括网关、缓存系统、数据库系统，也对应三种类型的服务器组，这三组服务器都支持多开，以下是系统架构图：



图2-1

**(一)数据结点的动态增加**

**1. 缓存动态增加节点**

我们的缓存系统是分布式的，支持动态增加缓存节点的，并且要实现读取缓存访问不到时，要先向持久化读取，再同步到缓存中，而当缓存节点增加时，必然会打乱之前的hash规则，造成数据访问不到，然而我们的缓存是支持使用持久化数据去重建已经被换出的缓存数据的，但重建的成本比较大，所以我们必须把这种重建成本降到最低，一致性哈希是一个非常好的选择，因它只影响第一个节点上的部分数据，而且我们的数据重建方式是有访问才会重建，所以对整体影响不大。

需要注意的是对于只存储缓存的数据，我们使用一致性哈希的方法增加节点后，有少量数据是不可恢复的。



图2-2

**2. 数据库动态增加节点**

数据库是持久化存储，也需要分布式存储，当需要动态增加存储节点时，如果用和缓存一样的一致性哈希，必然导致一些数据无法访问，这样就造成了数据丢失，所以数据库部分必须增加路由服务器，来存储持久化的key节点分布。

由于路由服务器对于持久化存储而言是至关重要的，所以路由服务器必须保证在路由服务器故障时有备份机代替其工作，这样我们的路由服务器就要支持备份功能，我们使用master-slaver的方式来做备份，slaver端采用复制master端的操作进行实时备份，而master端只发命令到slaver端，不等待回复，也就是说使用异步备份的方式进行，这样可以保证备份地效率，不影响本身提供的路由服务，一旦master出现故障，客户端自动转到某个slaver上，这个slaver就变成了master。

如下图所示：



图2-3

**(二)数据结构的设计**

**1.基本key-value类型**

key-value类型的数据字段说明：

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 意义 |
| key | 字符串 | 数据的键 |
| value | 二进制 | 非可见，客户端创建 |

<1>在redis中，原生支持key-value类型，value类型可以是redis叙述的string类型，string类型也包含了二进制类型。这里需要注意的是由于客户端对缓存的存取是并发有可能会存在数据的一致性问题，也就是说两次请求同时操作一个key的值，造成并发数据时造成的数据错误。

在我们的这样的并发系统里，很难去用事务、或CAS去保证数据的一致性访问，这要求我们的项目中，不能出现同时操作对同一数据进行赋值的情况，这种赋值必须是顺序的，不能是并发的。这里主要是体现在需求分析中的用户数据中，这里的数据改变都应该是用户主动请求的进行改变，而不能由其他用户触发，举例说明，如果一个用户A同时和两个用户B、C进行战斗，B、C同时结束了这场战斗，A在从两场战斗获得经验时，必须是A主动收到了结果后的主动操作，而不能是B、C结束这场战斗时去改变A的经验值，否则是会有数据一致性的风险的。

<2>在mysql中，没有原生的key-value类型支持，但mysql支持关系表，我们建立一张二维表就可以表示出这个关系了，但由于key-value数据本身主要是为了提高查询效率而使用的数据格式，我们用一张大的二维表，这显然是大大降低了原本的意义。我们按我们单个数据库的数据量计算出需要的数据条数，建立多张key-value表，按使用 hash算法把key散列到某张key-value表中，这样就大大提高了查询效率。

数据库需要注意的一个问题是如果一旦系统重启，或路由重启，我们需要建立路由表，而建立这张路由表的依据就是所有数据库中的key-value表，所以我们要在系统启动时读取所有key-value表，把有key对应的数据库结点位置注册到路由服务器中。

**2.计数器类型**

计数器类型主要是用于id分配等操作，需要支持的时自增或自减，我们把功能扩展下，支持定义自增量、自减量、清零的操作。

<1>在redis中，原生的支持了string类型的自增、自减、增量自定义、设置等等操作，这些都是原子操作，要求string类型必须是一个整形的字符串形式。

<2>在mysql中，我们需要建立一张二维表：

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 意义 |
| key | 字符串 | 适应系统的key-value数据 |
| value | 整形 | 值 |

这张二维表应该是静态的，每次有计数器类型的请求时，对应一个存储过程，从这个表中查找key，进行自增、自减等等操作，并把值返回。

**3. 列表类型**

列表类型要支持push、pop、delete的操作，注意这个列表的顺序应该是像队列一样先进先出的。

<1>在redis中，原生支持了list类型的数据结构，他是由一个key对应着一个list表，这个表可以支持插入、删除、查找，返回元素等多种接口，这些接口都是原子操作。

<2>在mysql中，要实现一个列表（队列规则），就需要建立一张二维表：

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 意义 |
| count | 整形 | 自动增长，保证先进先出 |
| element | 二进制 | 非可见，客户端创建 |

这张二维表的count字段是用来自动增长，在有元素插入后，整个表还可以保持一个队列的规则，在需要从表中拿出元素时，保证每次拿出的是count最小的值，这样做的目的是为了和redis保持同步。

**4.有序表类型**

有序表要支持插入、删除、排序、查询区间排序结果等操作。

<1>在redis中，原生的支持了zset类型的数据结构，他是由一个key对应着一个排序表，这个表支持所有我们需要的接口，且都是原子操作。

<2>在mysql中，要实现排序表，需要建立一张二维表，要根据某一列进行排序：

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 意义 |
| ranking-key | 字符串 | 用于排序的关键字 |
| element | 二进制 | 非可见，客户端创建 |

这个二维表中的element字段是不能重复的，所以我们要把element设置为主键，ranking-key字段是用来排序的关键字，但如果ranking-key相同，我们需要按照element来排序，因为redis中如果score是重复的，redis会按value来排序，所以我们要用一样的规则。这张二维表需要和我们系统的key-value基本数据对应起来，这就要求我们动态的建立这张二维表，并使用key值来命名这个表。

**(三)问题分析与设计调整**

（二）中四类数据结构的设计已经可以满足现有系统中的数据要求，我们的设计中需要保证缓存与数据库数据的同步，需要在系统出现一些故障时，进行子节点重启或子系统重启，所以我们需要保证：

**1.管理数据库中二维表。**

因为四种数据结构，我们建立了多种二维表，不对这些表进行管理，必然无法实现重新构建某些子系统，如路由系统，路由系统是key与数据库的对应关系，如果重启路由系统，需要根据key来构造路由表，没有对二维表的管理，就无法构建这个路由表，所以我们对二维表做以下管理：

<1>key-value基本表。用于记录key-value基本数据，这类表在数据库中是静态建立的，表的个数固定，暂时定为500个，表名：key-value-001~key-value-500。

<2>计数器数据表。用于记录计数器的key-value存储，这类数据相对比较少，数据表应该是静态建立的，平均到每个库中只需要一个表即可，表名：count-001。

<3>列表数据表。用作建立一个队列，每一个key对应一个，这类表是动态建立的，需要用第三方表来记录这些动态表，所以建立一个静态表，表名为summary-queue-001，每生成一个队列表，用key创建一条记录，插入summary-queue-001中。

<4>排序数据表。用作建立一个排序表，每一个key对应一个，这类表是动态建立的，需要用第三方表来记录这些动态表，所以建立一个静态表，表名为summary-map-001，每生成一个排序表，用key创建一条记录，插入summary-map-001中。

**2.数据写入顺序问题。**

在进行数据写入时，如果先写入缓存，再写数据库，一旦数据库写入出错，要恢复缓存是一件比较危险的操作，容易使实现复杂化，所以我们需要注意的是，一定要先写数据库，再同步的写入缓存。

**3.避免出现同一个key出现并发写入。**

虽然数据库和缓存都可以保证原子操作，但因为我们使用了缓存系统，客户端有了访问请求，我们需要先改变数据库，返回后同步缓存，如果缓存不存在这个key，就要用数据库的数据写入缓存，这时如果又有另一个相同的操作，这样的并发操作是危险的，如果顺序颠倒，缓存中的值就是错误的。

在我们之前设计的架构中并发问题尤其明显：图2-1中如果两个客户端的请求是对某key对应value的自增操作，而这个key在缓存中是不存在的（由于LRU被换出），请求到了gateway，先到的请求通过disk store system对数据库进行了加1操作，然后返回了其值（假设是1），然后返回给gateway同步缓存，这时第二个自增请求通过gateway到了disk store system，数据库中变成了2，然后返回给gateway同步缓存。由于gateway是分布式的，可能存在这两次请求分布在两台gateway上，而来自由cache store system返回请求后（cache中不存在，需要从数据库中同步到缓存），很难说哪台gateway先收到返回，也很难说哪台gateway先向cache store system发起缓存同步的消息，如果这个顺序是相反的，那么缓存里的数据可能先被设置为2，再被设置为1，于是就出错了。

为了解决这个并发问题，需要更改原有的系统架构，避免数据进行缓存与数据库中同步时，出现并发操。我们需要把这个操作变成顺序的，来解决这个问题，参考下图：



图2-4

原有架构是用一个mysql代理节点来操作一组mysql服务器，用一个redis代理节点来操作一组redis缓存服务器，而重新设计后的架构，把这部分进行了合并，用一个数据代理节点来操作指向相同key的一组mysql服务器与一组redis服务器，这样网关节点就不用关心数据在缓存和数据库中的同步问题，这个问题都交给了同一个数据代理服务器上进行，这个同步操作在一台服务器上进行就大大简化了这个问题。

数据代理服务器需要操作多台mysql和多台redis，如果使用单线程，必然是阻塞式的，这样就会降低系统的并发能力，所以这参数据代理节点也必须是多线程的，然而多线程本身也是并发操作，这也会引起缓存和数据库同步时的并发问题，我们也需要解决这个问题，最简单的解决办法就是按key划分到不同的线程，这样同一个key的处理必然在同一个线程中进行，就不会有并发问题，但这种方式在很多情况下，本来只需要读取缓存就可以返回的请求可能会因为前面有多个写请求而处理等待状态，降低数据读取的速度。对于这个问题，我们可以利用读写分离的方式进行，线程要成读线程和写线程，相同key的读操作在同一个读线程中进行，相同key的写操作在同一个写线程中进行，当读操作在缓存中没有读时，先向对应的写线程发送一条同步key的消息，然后向数据库读取后，直接返回；而写线程要处理网关发来的写请求，也要处理读线程发来的同步缓存的请求。这样的分配方式使大大增加了读操作的效率，但同时也会造成读取和写入操作会发生乱序的问题，也就是说两个客户端A和B，A请求一个写操作，然后B请求一个读操作，可能实际处理上是先处理读操作，再处理写操作的，如果有同一个客户端发出在同一次连接中连续发出两次请求，也可能造成这个问题，但由于我们本身存在多网关，由于不同TCP连接中包是没有顺序的，所以这个问题是不可避免的，而对于同一个客户端，我们一般是要求一次请求，等待返回后才能进行第二个请求，所以这个问题可以忽略。

以下是数据代理服务器中线程处理的流程图：



图2-5

**4.服务器节点的连接管理。**

因为我们的架构中有多种不同功能的服务器，每种功能的服务器可能都有多个节点，他们之前的连接关系也是非常复杂的（因为存在多节点与多节点之间的多对多的连接关系），这样每增加一台服务器节点，每减少一台服务器节点，其他相关服务器都应该动态感知，做出响应，而不是由配置和人工操作来进行，所以我们必须增加服务器管理器。

一般的多服务器节点的架构中一般会有一个中心节点，用来管理服务器关系，每增加一台功能节点，就会先向这台服务器注册，然后这台中心服务器会通知有关的其它节点，以这种方式来实现一些服务器节点的动态开关功能，而在我们的服务器架构设计中，是不允许出现这种热点的，因为一旦热点出故障，整个系统都会受影响，所以我们的服务器管理器一定应该也是分布式的，至少有备份机策略，以下是服务器管理器系统的设计图：



图2-6

使用中心服务器来管理其它应用服务器，当有服务器登录或退出时，用于通知与之相关联的应用服务器，在我们的设计中，把中心服务器改成了多节点的环形设计，用于解决热点故障问题 。

**三、具体设计与实现**

我们在设计过程中已经进行了架构设计和子系统的划分，整个系统被划分为以下几个子服务器系统：

* 网关服务器。实现为服务集群，负载平衡，对路由和数据代理的访问要求是非阻塞的。
* 路由系统。实现为多服务器，也就是一个分布式的hash表，预计单台进程的数据量在500万到1000万左右，占用内存约为1G左右，网关对路由的定位方法应该是按key的hash值取模，平均散列到多台路由机上。
* 数据代理。实现为服务集群，把整个数据代理子系统划分为多个更小规模C/S结构服务器，数据代理节点对redis和mysql进行阻塞式存取。

由于整个系统比较复杂，需要支持动态增加各种节点，各种容错，还有部分系统的重启等等一系列问题，所以我们在实现这个系统时，采用增量模型的开发方式，先实现整体的架构和客户端需要的接口功能，然后再实现动态增加节点，最后实现容错机制。

**(一)第一阶段**

第一阶段的目标是实现系统的整理架构与客户端需要的功能和接口。

**1.网关服务器的设计**

网关服务器除了负载平衡以后，最重要的工作是路由查询和数据请求，而这两个服务分别由路由服务器和数据查询服务器承担，所以这里的路由查询和数据请求只是向目标服务器发送消息，由于我们的方案要支持高并发，所以这的操作必须是非阻塞的，由此分析，我们设计成一个非阻塞的网关服务器。



图3-1

上图中用数字标记出了一条来自客户端的数据请求，经过路由查询、数据请求返回给客户端查询结果。

**2.路由系统的设计**

在第一阶段的实现中，主要是功能和架构的实现，而路由系统是redis和mysql服务器动态增加的重要依赖，而这一功能是第二阶段的主要任务，所以在第一阶段，不实现节点动态增加的功能，但路由系统还是要存在于整个系统，而且要是多点构架，所以我们需要实现的是：按key的哈希值定义到一台路由节点上，在这个路由服务器是进行哈希取模，定位到某一台数据访问代理服务器节点。

**3.数据访问代理的设计**

数据访问代理的设计前文已经做了一些叙述，由于数据访问代理服务器的使用线程的读写分离来加速读取速度，所以这里需要几个队列：

<1>读取请求队列。由网络线程接到读取请求后，把请求写入该队列，读取线程从该队列中取得任务，进行数据的读取工作。

<2>写入请求队列。由网张线程接到写入请求后，把请求写入该队列，写入线程从该队列中取得任务，进行数据的写入工作。

<3>同步缓存请求队列。在处理某个请求时，可能会出现由于redis中数据因LRU算法被抛弃掉后没有数据的情况，这时需要写入线程进行同步操作，所以就有了这个队列，每当读取或写入线程发现有这种情况时，把一个同步请求写入这个队列，等待写入线程下次从队列中取出请求进行同步操作。

由于写入线程在其线程循环中需要从两个队列中取得请求，这就要求我们在这个生产者消费者型中把两个队列共享同一个条件变量，这里的实现方法可以是两种，第一合并这两个队列，第二把这两个队列的封装起来同共享条件变量。

以下给出详细的内部设计方案：



图3-2

图3-2中，所有红色的过程都是存取redis和mysql的操作，这些操作全都是阻塞式的，这样做的目的，一是为了数据的一致性，二是为了降低实现的复杂度。

**4.管理服务器的设计**

管理服务器也就是中心服务器，由于我们中心服务器采用了多结点的环形设计，所以可以实现节点的动态增减，而其具体设计已在前文有所叙述（详见图2-6），在这里就不再做过多陈述。

**5.交互接口的设计**

(二)第二阶段

第二阶段的目标是实现系统主要结点的动态性增加。

(三)第三阶段

第三阶段的目标是容灾与备份。在缓存与数据库出现故障时，能够不影响系统运行，备份机器自动加入到系统中。