# 西游记数据存储方案

大型多人在线网络游戏(MMORPG)管理着庞大的用户数据, 由于游戏需求变化非常快,而且不像传统软件项目那样比较有规律可循, 加上游戏的版本迭代周期很短(平均2周)，所以对游戏数据的管理是一项极大的挑战。

大多数游戏采用的存储方案主要有两种：一种是通过大量的字段（int,char,datetime）等逐一表示游戏内容,如：

|  |  |  |
| --- | --- | --- |
| Name | Type | Length |
| role\_id | Int | 11 |
| role\_hp | Int | 11 |
| role\_mp | Int | 11 |
| role\_strength | Int | 11 |
| role\_intelligence | Int | 11 |
| role\_level | Int | 11 |

这种表示法的优点是字段意义明确，方便查询，修改，统计分析；缺点是研发过程中需要频繁的增删字段，修改字段类型。

另一种是通过几个blob字段来存储大对象数据，如：

|  |  |  |
| --- | --- | --- |
| Name | Type | Length |
| role\_id | Int | 11 |
| role\_property | blob | 0 |
| role\_task | blob | 0 |

当然这种方式的优缺点和第一种方法相反。

如果项目组成员都对数据库事务等高级用法非常熟悉，又有一套比较好的机制来保证数据的一致性，项目早期把各功能模块抽象的较好，需求变动小，那么用第一种无疑是最好的。

但是实际上很多项目做不到那么好的规划，经常临时改需求，加上游戏开发周期比较长（2年），当初的设计可能有很多已经过时，所以最终的需求往往和既定需求大相径庭，加上游戏逻辑非常复杂，又要保证事务，这时用blob来存储就会对程序冲击小一些，对开发人员要求也相对会低一些。

《西游记》根据我们项目组成员的实际情况，最终采用的是blob方式存储，上面简单介绍了blob存储的优缺点，实际项目中仅版本管理遇到的情况可能就很复杂，如：开发服务器，体验服，正式服（单区），正式服（全区），越南版，台湾版，马来西亚版等等，由于上线时间的差异，每种服务器的数据版本都可能不一样，如果要人工去维护这种差异，工作量和风险无疑都是巨大的。

传统blob是纯二进制，其中有一个版本号标志，只有指定的解析程序才知道如何去解释这些纯二进制数据，根据blob中不同的版本号调用对应的解析函数，下面这段代码就是这种方法很常见的做法，而且随着版本的升级，就会积累越来越多丑陋代码。

|  |
| --- |
| if (version == 3) {    ...  } else if (version > 4) {    if (version == 5) {      ...    }    ...  } |

为了彻底摆脱这种版本差异的束缚，让开发人员专注在研发上，需要有一种自动兼容各种版本数据的方案。也就是至少要用一种带标签和长度的表示法用来描述二进制数据，可能最先想到的就是xml，但是只要测试就会发现xml在占用空间和效率上太差强人意，在注重效率和空间的存储方案的还是不合适。

一次偶然的机会，接触到了Google Protobuf，发现设计之初就是用来解决不同版本协议兼容性的，而且在占用空间和效率上都要比xml优秀的多，下面是protobuf官方的比较数据：

|  |
| --- |
| * are simpler * are 3 to 10 times smaller * are 20 to 100 times faster * are less ambiguous * generate data access classes that are easier to use programmatically |

仅仅有了TLV ( tag/length/value) 类型的描述语言还是不够，要做到完全的自动兼容，让开发人员只关注最新程序，还是要做大量其他工作的。

一：　建立和游戏逻辑数据结构对应的PB存储结构

|  |  |
| --- | --- |
| **class Property**  {  Int role\_ strength;  Int role\_ intelligence;  Int role\_ spirit;  } | **message PBProperty**  {  Int role\_ strength = 1;  Int role\_ intelligence =2 ;  Int role\_ spirit =3 ;  } |

二：编写逻辑结构到PB存储结构的Get/Set函数

|  |
| --- |
| // 用PB数据给逻辑数据赋值  void Property：：GetFromPB(PBProperty\* pbProperty)  {  role\_ strength = pbProperty->role\_ strength();  role\_ intelligence = pbProperty->role\_ intelligence();  role\_ spirit = pbProperty->role\_ spirit();  }  // 用逻辑数据给PB数据赋值  void Property：：SetIntoPB(PBProperty\* pbProperty)  {  pbProperty->set\_role\_ strength(role\_ strength);  pbProperty->set\_role\_ intelligence(role\_ intelligence);  pbProperty->set\_role\_ spirit(role\_ spirit);  } |

三：数据存/取盘(序列/反序列化)

|  |
| --- |
| DB Blob  Binary  Message  PBProperty  Class Property |

四：数据升级实例

1. 增加字段（顺序向后增加）

|  |  |
| --- | --- |
| **class Property**  {  Int role\_ strength;  Int role\_ intelligence;  Int role\_ spirit;  Int role\_ endurance;  } | **message PBProperty**  {  Int role\_ strength = 1;  Int role\_ intelligence =2 ;  Int role\_ spirit =3 ;  Int role\_ endurance = 4 ;  } |

|  |
| --- |
| void Property：：GetFromPB(PBProperty\* pbProperty)  {  … // 保持前面数据不变  role\_ endurance = pbProperty->role\_ endurance();  }  void Property：：SetIntoPB(PBProperty\* pbProperty)  {  … // 保持前面数据不变  pbProperty->role\_ endurance(role\_ endurance);  } |

1. 删除字段（保持原有PB字段及索引不变）

|  |  |
| --- | --- |
| **class Property**  {  Int role\_ strength;  Int role\_ intelligence;  ***~~Int role\_ spirit;~~***  Int role\_ endurance;  } | **message PBProperty**  {  Int role\_ strength = 1;  Int role\_ intelligence =2 ;  Int role\_ spirit =3 ;  Int role\_ endurance = 4 ;  } |

|  |
| --- |
| void Property：：GetFromPB(PBProperty\* pbProperty)  {  role\_ strength = pbProperty->role\_ strength();  role\_ intelligence = pbProperty->role\_ intelligence();  ~~role\_ spirit = pbProperty->role\_ spirit();~~  role\_ endurance = pbProperty->role\_ endurance();  }  void Property：：SetIntoPB(PBProperty\* pbProperty)  {  pbProperty->set\_role\_ strength(role\_ strength);  pbProperty->set\_role\_ intelligence(role\_ intelligence);  ~~pbProperty->set\_role\_ spirit(role\_ spirit);~~  pbProperty->role\_ endurance(role\_ endurance);  } |

3.修改字段属性（类型，大小）

不能直接修改数据类型，需要通过走删除，增加的方式达到修改字段目的。

总结：

利用Protobuf的特性使用这个方案可以完美达到各种版本数据兼容的目的，其实相当于我们是在逻辑数据和数据表字段之间加了一个中间件，数据的存储和读取都通过这个中间件来完成，而这个中间件又非常的智能，可以自动的去识别各种版本的数据。

|  |
| --- |
| GameLogic |
| GoogleProtobuf |
| Mysql |

  **Protobuf**

参考<http://code.google.com/p/protobuf/>，里面有非常详细的文档说明和用例。