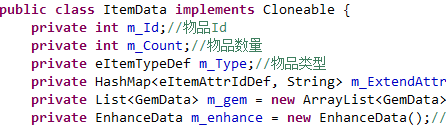
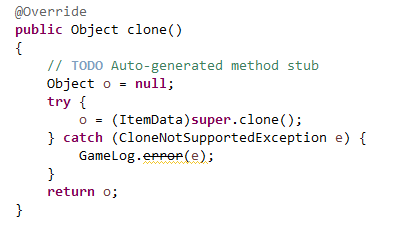
逻辑部分

1.clone方法实现错误

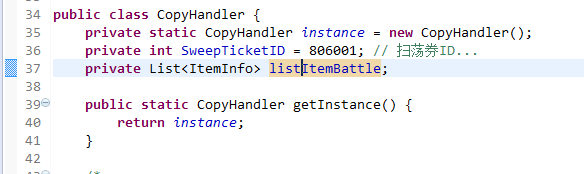




所有集合类，对象只是clone了引用，导致被clone的ItemData和clone后的多个ItemData持有相同一个m\_ExtendAttr、m\_gem，当出现修改操作就会造成灾难！因为修改其中一份等同于修改同一个母体clone出来的每一分。

2.客户端发过来的数据，服务器没有作验证，客户端发什么服务器就设置什么，比如镶嵌宝石，只要客户端作弊，可以直接弄个100W攻击力的佣兵出来



3. 针对玩家一一对应存储的数据居然放到XXXhandler对象上，游戏世界只有一份，多个玩家一起操作同一份互相替换，多人同时操作必然是灾难。这里的listItemBattle存储了玩家战斗后的掉落道具集，如果多个玩家一起战斗就彻底乱套了，低等级玩家拿了高等级玩家的奖励，高等级玩家找不到自己的奖励。除了副本还发现聊天模块也是如此，需要彻底检查所有模块

4.避免创建无谓的对象，如游戏逻辑中的各种Comparator

(1)RankingMgr

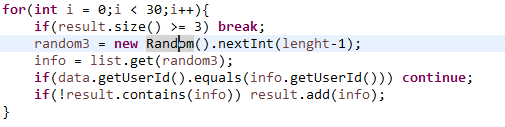


里面的RankLevel、RankFight、RankTeamFight可以作为RankingSort的静态变量存在

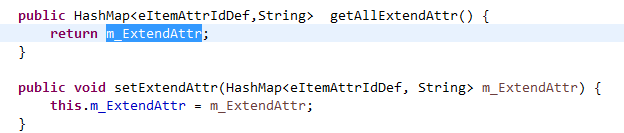
(2)OtherRoleHandler



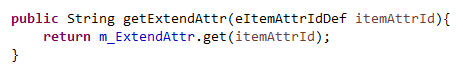
(3)循环中创建Random对象



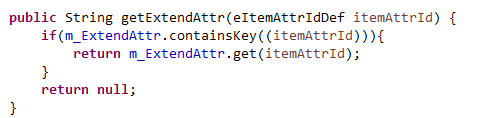
5.内部使用的Map不能直接返回给外部使用，如ItemData中的m\_ExtendAttr



更不能提供set方法，这样封装性完全没了！把外部会用到的功能用方法展现

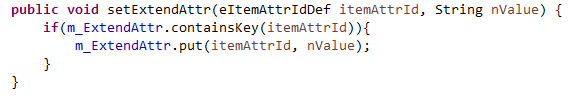


6.不要调用多余的方法

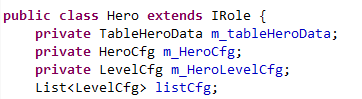


这个containsKey判断没有意义

7.像这样有内部条件的判断，需要显式的返回值告诉调用者成功或者失败(或者返回失败原因的枚举)

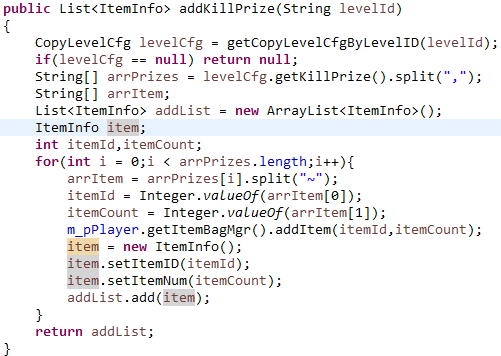


8.数据对象不应该直接持有配置类的对象 ，如Hero中的

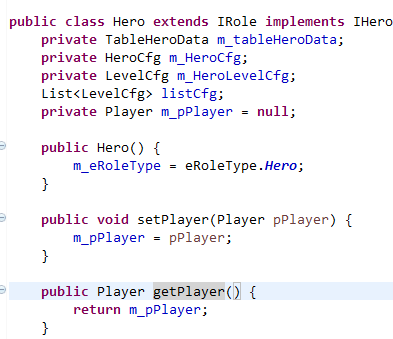


应该在逻辑使用Config的时候再根据 Hero的当前条件去获取，否则做热更新非常麻烦，而且也减少出错的概率

9.在循环中外部定义的作用域在效率上没有优化，使可读性变差了，谨记一个原则，最小作用域

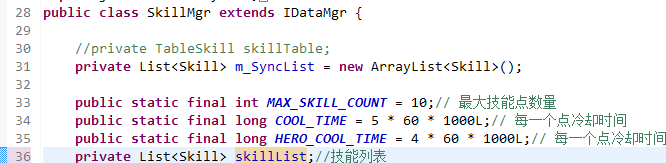


10.不合理的相互引用设计

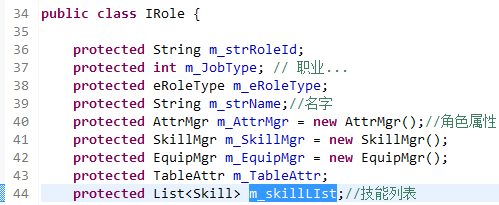


Hero不应该有返回Player的方法，更不应该持有Player对象，这里合理的做法是持有playerId

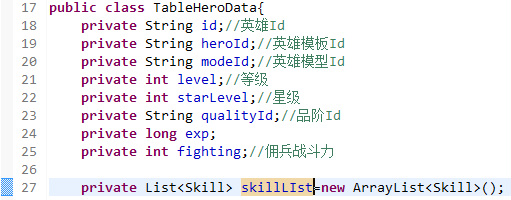
11.应该唯一表示一个概念的数据结构按为了使用方便存放在多个地方，如技能列表(List<Skill>)，这针对武将(或主角)必须是唯一的



SkillMgr持有一个技能列表，然后IRole在持有SkillMgr的同时自己额外在保存一份技能列表(m\_skillList)，在初始化的时候把SkillMgr的SkillList赋值过来，等于IRole与SkillMgr持有的是同一个对象，这是一个非常糟糕的设计，本来IRole持有SkillMgr，就可以获取SkillMgr里面的SkillList，完全没必要自己再存一份！

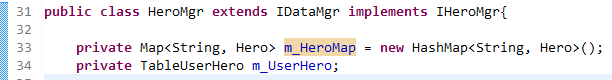


更严重的是Hero继承IRole，两份技能列表都能拿到，然后为了存储到数据库方便又再搞了一个TableHeroData来存技能列表，一个技能列表居然存了3个地方。

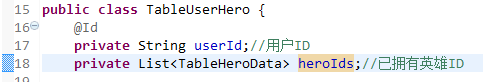




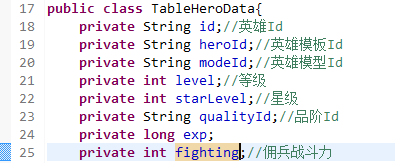
12. 上一个问题的延续版，HeroMgr持有Hero列表，每个Hero里已经有TableHeroData，但HeroMgr再搞了一个List把每个Hero里面T的ableHeroData又存了次，单纯为了调用起来方便一点点，把一个简单的数据结构彻底复杂化，不但对象的生命周期管理非常混乱，跟踪和定位问题的难度也指数型增加，毫无设计可言，最大的问题是在开发阶段，不觉得这些问题是问题，这些的数据设计到处都是，现在要改非常麻烦



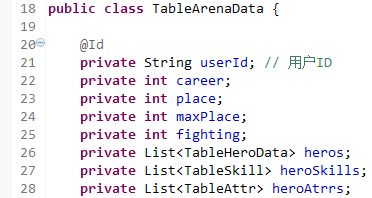




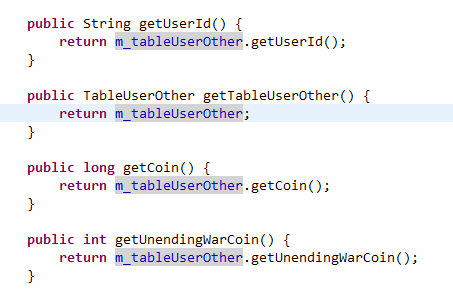
13.在运行中通过公式和配置计算所得的属性不应该保存到数据库，应只存内存，如佣兵的战斗力



14.用户相关的Hero数据、Skill数据应该只存在用户里面就ok，竞技场必须用到的话应该只保存id(heroId列表、skillId列表)，不能把hero对象拿到竞技场这里存放



15.重复的方法在正常情况下不要写多次，类似下面的m\_tableUserOther本身会返回出去，m\_tableUserOther中的方法就没必要在Player中又封装一次



多线程部分

服务器跑在多线程环境下，所有的数据都面临着并发安全问题。当前的逻辑代码编写没有做多线程保护，并且每一句逻辑都是假设在前一句逻辑能正确执行的情况下。

服务器中多线程可能引起的问题：

1. 数据丢失
2. 抛出运行时异常导致代码无法正常执行
3. 逻辑中读取到不正确的数据，导致逻辑无法正常执行
4. 死循环、死锁(暂时没有用到显式锁)

简单说是无法保证数据的安全性和服务器的稳定性！

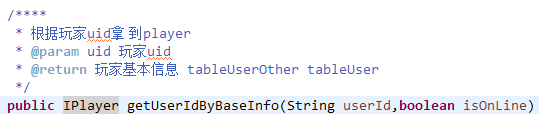
因此我们必须建立在一个多线程安全的模型上运行，首先我们理解一个概念

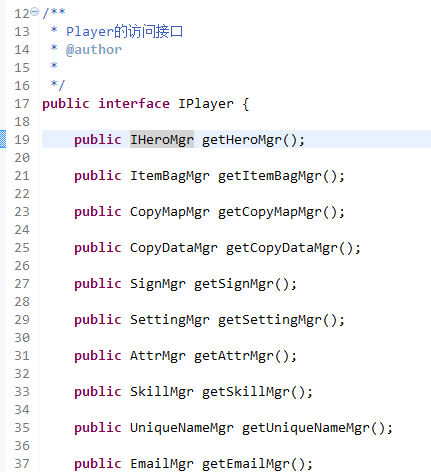
对于所有数据，我们只有一条线程写，同时可能有多条线程读，即写是单线程，读是多线程的，但读写之间不互斥(这与读写锁概率不一样)

第一步：列出哪些是玩家个人数据，只有玩家自己才会修改：比如玩家的背包数据，玩家的佣兵；哪些是公共数据，可能被不同玩家同时修改：比如排行榜、公会、竞技场

第二步：对玩家数据与公共数据耦合在一起的部分进行分析和分拆，比如公会数据在Player中有一份记录，在公会中也有记录，需要把在玩家的部分全部整理到公会中去。

第三步：对已有数据类做一层只读接口，让其他玩家访问(自己依然是访问原来的数据类，从重要的数据开始做起)

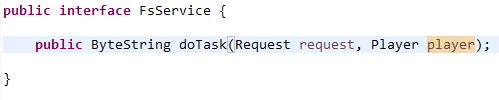




通过id拿到IPlayer接口，对Player只有读的权限。从IPlayer获取到其他数据类的接口也是只有读的权限，如IHeroMgr



玩家自己的Player通过doBusinessService的方法参数中传递，每个消息处理都能拿到自己的Player，但对于其他玩家，只能拿到IPlayer



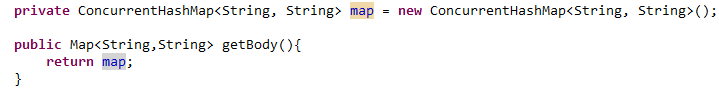
第四步：对集合类操作权限的限制

1.对于List类的只读操作统一返回List的拷贝，使用return new ArrayList<>(list);

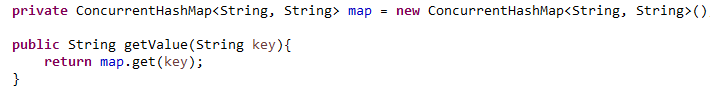
因为在多线程读单线程写的模型里面，对list进行迭代，遍历的时候有可能抛出IndexOutOfBoundsException、ConcurrentModificationException，导致无法往下执行，假如是一个不可逆的操作，将会导致严重的后果(比如一个先扣钱再给道具的操作，扣了钱之后抛出异常，给道具的操作木有执行)

2.对于HashMap在多线程读写的环境下统一替换成ConcurrentHashMap(确定是只读不写的情况可以继续使用HashMap，比如配置表)

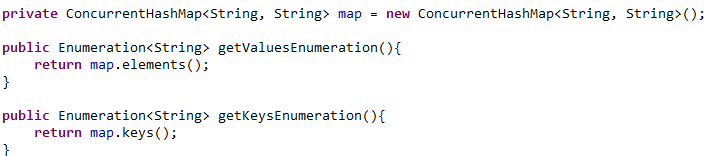
禁止这种写法：



大部分情况下可以用get方法代替



如需要迭代可以使用如下写法：



注意：ConcurrentHashMap不允许null键与null值，和HashMap不一样