ChatGPT

当前武器系统存在的问题

通过对仓库 Assets/Scripts/Weapons 下的武器系统脚本及相关 Player 逻辑的分析，发现当前设计在多方面存在不足。以下按照问题类别归纳，并在表格中列出主要问题点：

问题类别	具体表现与影响
职责划分不清晰	PlayerWeaponController 同时负责**玩家输入处理、武器列表管理、武器切换、射击触发**等多种功能，导致类职责过于宽泛。例如，它既订阅/轮询输入状态又调用当前武器开火 [1] [2]。此外，网络RPC最初也由该类处理，后来才部分迁移到 PlayerStatusManager，当前类中仍残留弃用的 RPC 方法 [3] [4] 和通过反射直接调用武器开火的黑魔法 [5]。这些迹象表明武器控制和输入/网络逻辑交织，存在“小神类”倾向。类似地，WeaponBase 基类囊括**弹药、冷却、后坐力、声音、事件**等诸多功能，也可能过于臃肿。
配置数据冗余重复	武器配置和投射物配置存在字段重复，增加了维护难度。例如弹跳次数和能量损耗既存于 WeaponData（MaxBounceCount/BounceEnergyLoss） [6] 又存于 ProjectileBase（_maxBounces/_bounceEnergyLoss） [7]。当前实现需要在 BouncyGun 中手动将 WeaponData 的弹跳参数传给投射物 [8]（见 SetBounceParameters 调用 [9]），增加了各类武器自行同步配置的负担。一旦修改弹跳逻辑，需要同时修改武器和投射物两处配置。类似地，伤害数值既定义在 WeaponData 又有投射物自己的 _damage 字段 [10]，网络生成时才通过 InstantiationData 同步 [11]；而本地生成未统一设置伤害，可能导致武器配置与投射物伤害不一致。再如 BouncyGun 蓄力射击临时尝试修改 WeaponData.ProjectileSpeed 来增加子弹速度（源码注释明确指出直接改 ScriptableObject 是不良实践） [12]，这都说明当前数据配置耦合度高、重复配置多，维护和动态调整代价大。
调用逻辑混乱低效	武器开火/换弹的调用流程存在一定混乱和重复。举例来说，WeaponBase.TryFire 内部会检查无法开火时自动调用 TryReload [13]，而 PlayerWeaponController 也在独立监听玩家换弹输入来触发 ReloadCurrentWeapon [14] ——换弹逻辑分散在武器和玩家控制器中，可能产生竞合。又如武器射击的网络同步，在 WeaponBase 中预留了虚拟的 NetworkFire RPC [15]，ProjectileWeapon 重写用于播放远端特效，但 PlayerWeaponController 曾经也直接RPC广播过射击指令（现已废弃） [3]。多处并行的网络调用机制增加了理解难度和出错风险。另外，PlayerWeaponController.ForceFire 方法通过反射私有方法直接调用武器 Fire [5] 绕过了冷却检查，这种做法违反单一职责和封装原则。**事件调用不一致**：WeaponBase 定义了 OnProjectileHit 静态事件但并未实际触发（StandardProjectile 中对此事件的调用被注释掉了 [16]），导致 WeaponUIManager 订阅了该事件却可能永远收不到命中通知 [17] [18]。这些现象反映出当前调用流程缺乏清晰结构，存在重复和不规范的部分。

问题类别	具体表现与影响
投射物逻辑耦合与膨胀	子弹/投射物系统在表现层实现上较为复杂，存在耦合和重复。ProjectileBase 承担了**运动、碰撞检测、弹跳、重力、生命周期、销毁、效果和声音**等全部逻辑，代码体量庞大 [19]。这导致：1）**耦合WeaponData**：投射物爆炸伤害半径等直接取自 WeaponData [20]，需要投射物保存来源武器引用 [21]；弹跳参数既来自自身字段又要由武器配置注入，耦合紧密。2）**逻辑重复**：StandardProjectile.ProcessHit 和 BouncyProjectile.ProcessHit 有相似的销毁判定逻辑，后者仅略微调整了弹跳优先级 [22]。碰撞处理 OnCollisionEnter/OnTriggerEnter 部分逻辑重复，两者都创建 RaycastHit 传递给 ProcessHit，但 Trigger情形下未处理销毁/弹跳 [23] [24]，实现不统一。3）**扩展困难**：投射物基类写死了很多行为（例如每次弹跳都生成音效对象 [25] [26]），若要引入新效果如追踪、自导引等需修改或继承大量代码。当前实现以继承方式提供了 StandardProjectile、BouncyProjectile 等，但增加新投射物种类（如散射弹片、地雷等）可能导致代码膨胀或功能重复。
武器切换与射击的网络同步	网络同步逻辑存在职责不清和结构不合理的问题。首先，武器射击/切换的同步职责曾经分散在PlayerWeaponController（已废弃RPC）和 PlayerStatusManager 中，所幸目前已经集中到后者，但 WeaponBase/ProjectileWeapon 仍然直接调用 PhotonNetwork 实例化投射物 [27] [28] 和 PhotonNetwork.Destroy 销毁投射物 [29] [30]。这种做法将网络细节耦合进武器和子弹类，违反了高内聚低耦合原则。一方面，每个投射物都是独立网络对象会产生较大开销，ProjectileManager 虽引入了批量同步优化 [31] [32] 和对象池设想，但武器代码仍直接用 PhotonNetwork.Instantiate，没有完全利用管理器统一管理。另一方面，不同武器可能需要不同同步策略（如Hitscan武器不生成投射物，只需广播射击结果），但当前架构缺少统一接口，各类武器各自处理网络导致代码分散。总体来看，网络同步部分职责边界不清，缺乏中心管控，难以保证一致性和效率。
扩展新武器类型困难	现有结构主要围绕射弹类远程武器设计，对未来扩展激光武器、近战武器、AOE武器支持不足。WeaponData 含有大量字段，但其中许多只适用于特定武器类型（如弹匣容量、弹药类型对近战无意义，引力引爆参数对普通枪支无用），**所有武器共享一个数据结构可能导致无用字段冗余**。对于激光/射线武器，目前无专门类，需新建类似 HitscanWeapon 处理瞬时射击逻辑，可能与 ProjectileWeapon 代码重复。近战武器虽可复用 WeaponBase 部分接口，但诸如开火频率、后坐力等概念并不适用，需要在实现中绕过或忽略，违背接口隔离原则。AOE武器（如手雷）可能需要延时爆炸，目前 StandardProjectile 只支持即时撞击爆炸或生命周期到时销毁（不带伤害），缺少定时引爆机制。可见，当前结构在添加新武器时缺乏灵活性，**没有明确定义不同武器类别的抽象接口或共同管理机制**，新增功能往往需要修改现有代码或大量复制粘贴，扩展成本较高。

以上问题表明，当前武器系统存在架构上的缺陷，包括模块职责划分欠佳、数据设计不合理、调用与同步逻辑混乱等。这些问题会导致系统维护困难、Bug 难以排查、扩展新功能阻力大。下面将结合上述问题，提出详细且可操作的重构方案。

武器系统重构方案

重构的目标是**解耦职责、消除重复、优化数据配置、引入通用工厂和管理器、规范网络同步**，并为未来扩展提供良好的基础。在不破坏现有 WeaponData 和 Projectile 框架的前提下，可以采取以下改进措施：

新架构概述

设计思路：将武器系统划分为清晰的模块层次，包括：武器逻辑抽象层、具体武器实现层、投射物与伤害处理层、管理器与辅助组件层。每层各司其职，通过事件或接口交互。总体架构如下：

- **WeaponBase**（武器基类，抽象类）： 保留通用属性（弹药、冷却、后坐力等）和方法接口（Equip/Unequip、TryFire/Reload 等），但精简自身逻辑，**不直接包含具体开火实现**。引入抽象的 `Fire()` 实现委托给子类或策略。仍负责触发武器事件（OnWeaponFired等）供UI和动画使用。

- **具体武器类**（WeaponBase 子类）: 根据武器类别细化职责：

- ProjectileWeapon（投射物武器）: 负责生成投射物子弹。重构为利用**投射物工厂/管理器**来创建或获取子弹对象，而非直接 PhotonNetwork.Instantiate。【改】其 Fire 实现中，通过调用例如 `ProjectileFactory.CreateProjectile(this, direction)` 完成子弹生成和发射，内部由工厂处理网络和对象池逻辑。这样 ProjectileWeapon 无需知晓 Photon 细节，实现与网络解耦。
- HitscanWeapon（命中扫描武器，如激光枪）: 新增类继承 WeaponBase，实现瞬时命中逻辑。在 Fire 中执行射线检测，对目标直接造成伤害。可调用通用的伤害处理接口（如 DamageSystem 或 IDamageable），并触发统一的命中事件（类似 ProjectileHit 事件）供其他系统使用。网络同步通过管理器广播射击结果（命中与否、伤害值等），远端播放激光效果。
- MeleeWeapon（近战武器）: 新增类继承 WeaponBase，实现近战攻击逻辑。在 Fire 中触发近身范围检测或动画事件，对范围内目标应用伤害。可不使用弹药系统（WeaponBase中MagazineSize可设为0表示无限），并屏蔽不适用的属性。近战攻击的网络同步可由管理器发送一次性攻击命令（包括攻击时刻、命中目标等）。

- （其他特殊武器类别可类似扩展，例如 AOEWeapon 用于抛射物/手雷，内部计时引爆。）

- **WeaponData 重构**: 保留通用字段，将特殊字段按武器类型归类管理：

- 可引入**子对象或子结构**表示投射物参数。例如在 WeaponData 中添加 `ProjectileConfig projectileConfig;`（或者直接引用对应的 Projectile prefab/scriptable object），其中包含投射物速度、生命时长、弹跳次数、爆炸半径等。ProjectileWeapon 加载配置时将这些参数传递给投射物生成。对于非投射物武器，该子配置可为空或忽略。
- 将不适用于所有武器的字段标记清晰或拆分。例如 InfiniteAmmo、MagazineSize 对近战无意义，可在逻辑上对 MeleeWeapon 忽略这些字段，或者通过 WeaponData 提供辅助属性（如 WeaponType 枚举）来判断适用性，避免误用。

- WeaponData 继续作为中心配置，但**杜绝运行时修改其字段**。如需动态调整（蓄力改变子弹速度等），通过参数传递或临时变量，不直接改 ScriptableObject。上文提到的蓄力修改速度问题，可改为 WeaponBase 保留一个运行时属性 CurrentProjectileSpeed，Fire 时根据_chargeRatio 计算后传给投射物，而不改 _weaponData.ProjectileSpeed。

- **ProjectileBase**（投射物基类）: 精简职责，侧重运动和碰撞检测，将伤害判定和效果触发委托出去：

- 去除直接调用 IDamageable 的逻辑，将其封装到**伤害处理模块**。例如，引入 `IDamageDealer` 接口或 DamageSystem 单例，由投射物在碰撞时调用 `DamageSystem.ApplyDamage(this, hitCollider, damageInfo)`，让DamageSystem负责查找IDamageable并应用伤害。这样投射物不直接依赖具体目标实现，更易扩展（例如以后可以让DamageSystem处理特殊护盾判定等）。
- **解耦 WeaponData**: 投射物不再频繁查找 WeaponData，而改为在生成时由工厂/武器传入必要参数。一种做法是在 ProjectileBase 增加统一的 Configure(config) 方法或属性赋值，例如 `projectile.SetParameters(damage, speed, maxBounce, explosionRadius, ... )`，由 ProjectileFactory 在生成后调用 [8]。BouncyProjectile 等子类可在 OnEnable/Start 时使用已配置的

参数初始化自己，从而避免每次撞击再通过 _sourceWeapon 去取 WeaponData。同理，引力效果 GravityForce、GravityRadius 可作为参数传入特定投射物（如果有“黑洞弹”等），而不是每个 ProjectileBase都挂载不使用的字段。

- **简化事件触发**: ProjectileBase 应正确触发 OnProjectileHit 事件，一旦命中处理完成（确定应当销毁或停止时）调用事件 [16]。此事件由 WeaponUIManager 等监听用于命中反馈。保证事件的单一来源和及时触发，清理之前遗漏或冗余的事件调用逻辑。

- **Projectile子类**: 保留 StandardProjectile 和 BouncyProjectile 等，实现各自特殊行为，但通过调用基类提供的扩展点减少重复：

- StandardProjectile: 专注于处理穿透和爆炸逻辑。可将爆炸计算封装到基类通用方法，如 ProjectileBase 提供 `Explode(damage, radius)` 实现AOE伤害，如需不同爆炸效果子类可重载覆盖。这样其他具备爆炸的子弹类型也能重用代码。

- BouncyProjectile: 仅扩展弹跳后的增速和特效，无需重复伤害计算逻辑。在新的架构中，它可以更多利用基类提供的弹跳事件或调用 DamageSystem，而不需要重写 ProcessHit 的大部分内容。其 SetBounceParameters 将被整合进通用 Configure 流程，不再由武器单独处理。

- **管理与辅助模块**:

- **ProjectileManager/Factory**: 引入专门的工厂类负责投射物生成、复用和销毁。例如 ProjectileManager 实现投射物对象池，预先缓存常用子弹类型一定数量，提供 `CreateProjectile(WeaponBase source, Vector3 position, Vector3 direction, float speed, float damage, ... )` 方法：如果有可用对象则复用，没有则实例化。无论本地或网络模式，武器类都通过该接口请求子弹。
    - 网络逻辑: ProjectileManager 持有 PhotonView，负责在本地生成投射物后，通过RPC将弹道参数广播给其他客户端。远端客户端的 ProjectileManager 接收到参数后，可以**本地生成一个可见但不具备碰撞的子弹对象**（纯视觉，用 TrailRenderer 等模拟轨迹），或者直接在UI层画出弹道。这与当前 ProjectileWeapon.NetworkFire 中的逻辑类似 [33]，但由中心管理器统一处理。这样可以避免每发射一次都生成PhotonNetwork.Instantiate，同时保证不同武器的网络同步采用一致方式。
    - 对象池: ProjectileManager 维护一个 Dictionary 或 Queue 存放闲置的Projectile对象，投射物销毁时不立刻Destroy而是回收。对于PhotonNetwork管理的对象，可采用“关闭GameObject并延迟销毁PhotonView”或使用Photon的对象池接口（若有）减少频繁分配。由于当前实现里PhotonNetwork.Destroy还是被调用了 [29], 重构后应尽量由ProjectileManager统一回收/销毁，以免遗漏。
    - ProjectileManager 还可统一处理**投射物ID**和**批量状态更新**（如当前已有的BatchUpdate思路 [31] [32]）。即由本地权威端周期性发送所有主动投射物的位置/速度，远端平滑更新已有对象位置，用于跨网络的同步校正。这种做法可取代每个投射物各自的PhotonView Transform同步，避免大量网络开销。
- **PlayerWeaponController 拆分/精简**: 将其重命名为 WeaponInventory 或在其内部**解耦输入处理**和**武器管理**两部分：
    - 输入处理部分可以移交给 PlayerInput 或一个独立的 WeaponInputHandler 脚本。由 PlayerInput 直接调用 WeaponController 的公共方法（TryFire、SwitchWeapon等），而不是 WeaponController内部去轮询输入 [1]。这减少WeaponController对Input的依赖，使其更关注武器状态管理本身。
    - 武器管理部分继续负责武器列表和切换，但应简化流程。切换武器时，仅处理本地状态和UI更新，由 PlayerStatusManager 或 NetworkManager 决定何时同步给其他客户端。这避免之前本地和网络多处重复调用。同样，TryFire 只检查并调用当前武器的 TryFire，不再掺杂网络RPC，改由武器开火后 PlayerStatusManager 捕获事件再广播。

- **PlayerStatusManager/NetworkManager**: 集中管理所有**玩家状态的网络同步**，包括武器装备和射击动作。重构方案中，可让武器的关键动作通过事件通知该管理器，再由其调用PunRPC：
  - 例如，本地玩家调用 CurrentWeapon.TryFire 成功后，WeaponBase.OnWeaponFired 事件触发，PlayerStatusManager收到后调用自身PunView.RPC通知其他客户端“玩家X使用武器Y射击了，方向D”。远端PlayerStatusManager接收RPC后，再调用对应远端玩家对象的CurrentWeapon.NetworkFire(direction,timestamp)，执行开火视觉效果[33]。这样，武器类本身不直接进行Photon网络调用，只关心自身逻辑，网络同步细节由管理器处理，实现**网络职责单一化**。
  - 武器切换同理，本地切换后通过 OnWeaponSwitched 事件或直接调用StatusManager的方法，广播给其他客户端执行同步切换。这其实在旧架构中已经开始这么做，只需确保PlayerWeaponController 不再直接RPC，而完全交由StatusManager处理。
  - PlayerStatusManager 还可以处理**武器动画同步**：当本地开始装弹或射击时，利用WeaponBase.OnReloadStarted、OnReloadCompleted 等事件，通过StatusManager RPC通知远端，由远端角色的动画控制器播放相应动画。这在现有系统中也已有雏形（WeaponData中定义了动画名称，StatusManager通过事件触发 PlayerAnimationController 播放[34][35]），重构时注意保持或完善这一机制。
- **DamageSystem/CombatManager**: 引入一个统一的伤害处理管理类，负责处理各种伤害来源（投射物、近战、爆炸等）对目标的作用。所有伤害事件通过此管理器来完成：
  - 继续沿用 IDamageable 接口，但由 DamageSystem 实现例如 `ApplyDamage(IDamageable target, DamageInfo info)` 方法，内部调用 target.TakeDamage并处理溅射、团队伤害等规则。ProjectileBase和HitscanWeapon在产生伤害时，将 DamageInfo 交给 DamageSystem，从而解耦对具体对象组件的直接依赖。如此可以在DamageSystem中集中处理诸如**爆头判定**（利用 WeaponData.CanHeadshot 等）或**伤害衰减**（激光距离衰减）等，更容易扩展全局战斗规则。
  - DamageSystem 也可统一管理命中事件：无论是投射物击中还是射线击中，都调用DamageSystem，进而由它触发全局 OnHit 事件。这样 WeaponUIManager 只需订阅一个伤害事件即可弹出命中标记，而不必分别订阅 ProjectileBase.OnProjectileHit 和未来Hitscan的事件。

上述架构以清晰的接口和管理器划分了领域逻辑：武器类只关心自身行为，通过工厂产生效果和通过事件报告状态；投射物只管物理运动和通知碰撞，伤害和销毁交由外部决定；玩家控制器不直接处理网络，同步由专职管理器完成。下面分模块详细说明改进要点：

1. 职责抽象与接口划分

明确划分各脚本职责，避免“上帝类”。核心做法是引入抽象接口和中间层来隔离不同功能：

- 定义 **IWeapon** 接口（或抽象基类 WeaponBase 已近似于接口），包含 `Equip()`, `Unequip()`, `TryFire(direction)`, `TryReload()` 等方法。所有具体武器实现该接口。必要时可进一步拆分接口，例如 IProjectileWeapon（带 ProjectilePrefab）、IHitscanWeapon 等，用于标识不同能力。但通常通过类继承结构区分更直观。

- **PlayerWeaponController**: 调整为只持有 IWeapon 列表并管理当前武器，不关心武器内部如何实现Fire/Reload。它通过接口与武器交互，实现对所有武器类型的统一管理。如将其更名为WeaponInventory，提供 `SelectWeapon(index)`, `GetCurrentWeapon()` 等方法。输入处理移出后，它仅响应高层次调用，提高内聚性。这样当将来增加新武器类型（比如近战），Inventory仍可照常管理，因为近战武器同样实现了 IWeapon，只是内部 Fire 行为不同。

- **输入逻辑**: PlayerInput 可以直接调用 WeaponInventory.TryFire 等，无需再在 Update 中反复查询，减少 PlayerWeaponController 的复杂度。比如 PlayerInput 检测到射击按键，则执行

`weaponInventory.TryFire()`；检测到换弹键则调用 `weaponInventory.ReloadCurrentWeapon()`。这一更改使输入响应更直接，也防止因 Update 顺序或状态标志导致的重复调用问题。

- **WeaponBase**: 继续作为抽象基类，实现 IWeapon接口默认方法，同时提供**钩子方法**供子类覆写：

- 抽象 `FireImplementation(direction)`：取代当前的 protected abstract Fire，用于真正执行射击。WeaponBase.TryFire 中除了一般检查外，不再直接调用子类 Fire，而是统一处理弹药扣减、冷却计时、事件触发等，然后调用 FireImplementation来产生效果。这样可以确保不管何种武器，都会正确执行公共流程。例如：

```
public virtual bool TryFire(Vector3 dir) {
  if(!CanFire) { ...return false; }
  bool result = FireImplementation(dir);
  if(result) {
    _lastFireTime = Time.time;
    if(!_weaponData.HasInfiniteAmmo) { _currentAmmo--; OnAmmoChanged?.Invoke(...); }
    OnWeaponFired?.Invoke(this);
  }
  return result;
}
protected abstract bool FireImplementation(Vector3 direction);
```

如此一来，具体武器只需关注生成子弹或造成伤害，弹药和事件由基类统一处理，避免重复编写、也防止遗漏事件。**（注：保持 OnWeaponFired 事件单一触发点，UIManager 则能可靠更新子弹数和开火提示）**。

- 针对　Reload/Equip等类似处理，也可采用这样的模板方法模式，将通用部分上移，差异部分下放。例如 Reload 时基类检查是否能装填和设置状态，随后调用子类的 ReloadImplementation（如果需要特殊动画等）。

- **HitscanWeapon / MeleeWeapon**: 作为 WeaponBase 子类实现 FireImplementation：

- HitscanWeapon.FireImplementation：利用射线检测目标，组装 DamageInfo 调用 DamageSystem。因为继续使用 WeaponBase.TryFire 框架，所以弹药、冷却以及事件都自动处理，无须担心武器类型不同而丢失逻辑。例如，WeaponData.FireRate 也适用于激光武器的开火频率限制，UI弹药也可显示“∞”表示无弹药限制。
- MeleeWeapon.FireImplementation：可以触发近战攻击动画或范围检测，同样在完成后，通过DamageSystem对范围内敌人ApplyDamage。由于近战通常没有弹药和冷却限制，可在WeaponData中配置为单发模式、InfiniteAmmo等，WeaponBase.CanFire逻辑自然会允许每次点击都可攻击（或可设FireRate限制攻击频率）。**（如果近战需要持续按键连招，可扩展WeaponBase支持Hold型触发等，但超出当前范围）**。
- 通过这种子类扩展，新武器类型融入体系而无需改动现有管理逻辑。例如WeaponInventory不关心当前武器是枪还是刀，只会调用 TryFire，逻辑由各自实现处理。

2. 数据配置与管理改进

消除冗余配置、统一管理武器与子弹参数：

- **WeaponData 与 ProjectileData**: 引入**投射物配置对象**以区分不同武器的数据需求。可以通过两种方式：

- 复合对象: 在 WeaponData 内增加一个 `ProjectileSettings` 字段（可为一个序列化类或 ScriptableObject 引用）。对于投射物武器，配置其弹丸速度、寿命、重力、弹跳、爆炸等参数；对非投射物武器，该字段可留空或默认。例如：

```
[Serializable] class ProjectileSettings {
  public float Speed;
  public float Lifetime;
  public int MaxBounce;
  public float BounceEnergyLoss;
  public float ExplosionRadius;
  public float ExplosionDamage;
  // ...etc
}
public ProjectileSettings projectileConfig;
```

  ProjectileWeapon 在开火时，如果 weaponData.projectileConfig 不为空，则将其中参数传递给生成的 ProjectileBase。**这样所有与子弹相关的参数统一由WeaponData管理**，不再散落在Projectile预制上，减少重复。需要改变弹跳次数或爆炸威力，只改 WeaponData 即可，保持单一数据源。
- 预制引用: WeaponData 直接引用一个 Projectile prefab（或脚本）和基础伤害等，由该预制上的 ProjectileBase 携带自身运动特性。WeaponData 提供 damage、projectilePrefab 字段，ProjectileBase 保留速度、重力等序列化字段。这其实与当前做法类似，但需要**建立一致性**：确保每种武器的 projectilePrefab 的属性和 WeaponData相匹配。重构时可以在 ValidateConfiguration 时检查同步，比如 WeaponData 中的Damage和prefab上的初始_damage一致，否则警告。这种方法简单但仍有配置分散问题，不如前一种集中配置方案。

- 综合考虑，**方案1**（ProjectileSettings子配置）更清晰：它避免了WeaponData包含无关字段，例如近战武器的 WeaponData 可以将 projectileConfig 置为空，编辑器看到的配置项就是空的，不会误配。而当前设计把所有字段都列在 WeaponData，不同武器通用/专用字段混杂，容易混淆。

- **参数传递**: 确保投射物生成时从 WeaponData 获取所有必要参数：

- ProjectileFactory.CreateProjectile 时，从武器的 WeaponData.projectileConfig 中读取 speed、damage、lifetime、弹跳等，传给 ProjectileBase.Configure。这样ProjectileBase内部不再需要自行从 _sourceWeapon.WeaponData 获取，相应减少 WeaponBase 引用的强耦合。
- 对于网络生成的子弹，同样通过RPC将这些参数发送，而不是仅发送 velocity和damage再由 StandardProjectile 去查 Weapon 11 。可发送 WeaponID 或 Weapon类型用于远端确定特效类型，但无需逐一字段查找。

- 例如，改为RPC传输序列化好的 ProjectileSettings struct，这样远端拿到直接用于配置自己生成的子弹对象，比当前 StandardProjectile.ConfigureFromNetworkData 简洁可靠。

- **避免重复维护**: 减少同一参数多处存储：

- 移除 ProjectileBase 中冗余的配置序列化字段，尤其是那些 WeaponData 已提供的。例如 _damage, _speed, _maxBounces 等可改为非序列化，仅作为运行时变量，来源于WeaponData配置。ProjectileBase本身只需少量通用属性（如引用刚体、当前状态等），其行为完全由传入参数决定。

- 这样做的好处是：修改武器配置立即作用于后续生成的所有子弹，无需再去Prefabs逐个调整。维护者在 WeaponData 面板可以直观地调整武器性能而不用切换到 Projectile预制去改重复字段。

- **特殊参数**: 对于不同武器特殊效果（如黑洞引力、燃烧DOT），可在 WeaponData 或 ProjectileSettings 中以选项开关和参数组合体现。如 WeaponData.HasGravityEffect [36] 可决定ProjectileBase是否应用CustomGravity。将来新增效果，也往此扩展，不直接写死在Projectile类里判断。通过数据驱动效果的启用，让拓展更方便。

- **动画配置**: WeaponData 已经包含 FireAnimationName、ReloadAnimationName 等动画标识。重构中应**继续保留并利用**这些字段：

- 在 PlayerStatusManager 或 AnimationController 中，订阅 WeaponBase.OnWeaponFired、OnReloadStarted 等事件时，根据 CurrentWeapon.WeaponData 的动画名调用 Animator。这样动画与武器解耦，添加新武器只需在配置中填写好动画状态名，逻辑层无需修改。【确保】WeaponData动画字段被正确使用，如前述 OnWeaponAnimationTriggered 机制。
- 若有复杂动画过渡，可考虑在 WeaponData 或另一个配置表中添加状态机信息，但这是拓展方向。

3. 武器与投射物解耦

降低武器与子弹实现间的直接耦合，通过工厂和事件机制连接两者：

- **投射物工厂模式**: 上文多次提到，通过 **ProjectileFactory/Manager** 实现武器开火对具体 Projectile 实例化的解耦。一旦引入这个中介，武器类无需知道**如何**创建子弹，只需要提出“我要在某位置朝某方向发射一颗子弹”的请求。工厂内部可根据 Weapon 提供的信息选择相应类型的投射物：
- 可以为每种武器预先注册其投射物 prefab（比如使用 WeaponData.weaponName 作为键），也可以从 WeaponData 字段直接拿引用。如 `ProjectileBase prefab = weapon.WeaponData.ProjectilePrefab;`，然后从对象池取出该类型对象。
- 工厂处理完成实例化/复用后，调用 projectile.Launch(direction, speed, sourceWeapon, sourcePlayer) 进行发射初始化 [37]。此时内部会调用 Rigidbody.velocity 等，不需要武器类操心。

- **好处**：将来如果换成另一种生成方式（比如基于地址ables异步加载子弹，或切换网络实现），只需修改工厂，不影响武器代码。这就是典型的OCP（开放封闭）原则应用。

- **弱化Weapon对Projectile子类的特定依赖**：当前 BouncyGun 直接将生成的 projectile 转型为 BouncyProjectile 来设置特有属性 [38] [8]。重构后尽量避免这种操作：

- 将这些特别配置通过**多态或接口**实现。例如，让 BouncyProjectile 重写 ProjectileBase.Configure，使其在基类配置基础上增加自己的trail颜色和弹跳参数，而工厂始终调用基类接口：

  ```
  ProjectileBase proj = Instantiate(prefab);
  proj.Configure(settings, sourceWeapon);
  // BouncyProjectile.Configure 内部检测到 settings 包含MaxBounce等则调用 SetBounceParameters
  ```

  这样，工厂不需要知道 proj 是哪种子类，更不需要Weapon去判断并调用特定方法。一切配置逻辑都封装在投射物自身的实现中。**遵循LSP（里氏替换）原则**：工厂处理的是ProjectileBase，不关心具体子类区别。

- 如果有极端特例，比如**充能武器**发射的子弹速度随蓄力不同，可以通过参数传递解决：蓄力比例由武器算出，通过 ProjectileSettings.extraMultiplier 传给 Configure，让 projectile 知道初速度需乘以此系数。如此仍不需直接在武器代码中改Projectile内部值。

- **伤害处理去中心化**: 先前提到将伤害计算移至 DamageSystem。这对于解耦武器和投射物也有帮助。WeaponData 定义的伤害参数如 HeadshotMultiplier、ExplosionDamage 等，不需由投射物自己读再算，而可在 DamageSystem应用：

- 例如DamageSystem.ApplyDamage时检查 damageInfo.damageType：
  - 如果是 Projectile 类型且目标是头部碰撞，则乘以 weapon.WeaponData.HeadshotMultiplier。【好处】是以后即使换成激光武器调用 ApplyDamage，同样会检查HeadshotMultiplier，只要WeaponData里配置，该机制就通用，而不用每个武器类/子弹类各自实现一遍爆头加成。
  - 爆炸伤害在 StandardProjectile.Explode 计算了距离衰减 [39] [40] ，这部分逻辑也可移到 DamageSystem：传入爆炸中心点、半径和基础伤害，由DamageSystem遍历范围目标并计算伤害。这样一方面 ProjectileBase 不用关心具体伤害算法，另一方面多个武器若有AOE效果可以共用DamageSystem的函数，不必复制代码。

- 总之，通过引入DamageSystem，武器/投射物只生成 DamageInfo（描述伤害来源、数值、类型等 [41] [42] ），具体应用由DamageSystem执行，实现**武器->伤害影响**的解耦。这也符合单一职责：武器管发射，DamageSystem管伤害结果。

- **事件通信**: 利用事件实现武器与投射物的松耦合协作。例如：

- OnProjectileDestroyed 事件（ProjectileBase提供）可由 Weapon或Manager监听，用于在子弹生命周期结束时执行后续逻辑（如统计命中数、特殊效果触发等），而不需要投射物直接调用武器的方法。当前ProjectileBase已有 OnProjectileDestroyed 事件 [43] ，应充分利用。
- OnWeaponFired 事件（WeaponBase提供）由ProjectileManager监听，收到后从Weapon获取必要数据生成投射物。事实上，可以彻底简化WeaponBase.FireImplementation为仅触发 OnWeaponFired(this, direction) 事件，其它都交给监听者完成。考虑到 WeaponUIManager 等也监听该事件更新UI [18] , 保持这一机制有利于解耦多方。
- 注意防止事件双重触发或错乱：比如之前 StandardProjectile 内曾尝试 OnProjectileHit?.Invoke 被注释 [16] , 重构中决定权在于 DamageSystem/ProjectileManager——当确认击中有效目标且造成伤害，就触发一次全局“Hit”事件，否则不触发。这比每颗子弹见物就报要严谨一些，具体规则可在 DamageSystem实现。

4. 引入工厂和对象池机制

通过工厂和对象池优化对象生命周期管理，提高性能并简化网络同步：

- **对象池实现**: ProjectileManager将管理一个或多个对象池，对每种Projectile类型维护队列：
- 在初始化时按需 Instantiate 一批隐藏的Projectile对象保存起来。可以考虑 pool大小由 WeaponData或配置决定（如普通子弹池100，火箭池10）。
- Weapon开火请求子弹时，如果池非空则取出对象，设置其位置和朝向，启用它；如果池空则 Instantiate新的。投射物脚本需在激活时重置必要状态（_currentBounces等计数、_isDestroyed标志等）。
- 当Projectile生命周期结束（Collision触发DestroyProjectile或超时），改为：如果Photon网络管理，则PhotonNetwork.Destroy(obj)通知各端，在Destroy回调中由各客户端ProjectileManager回收到池；如果是本地模拟对象，则直接SetActive(false)并回收到池列表。当前DestroyProjectile逻辑需调整：本地owner不直接Destroy(gameObject)，而是通知管理器回收；远端调用NetworkDestroy时同理。
- 需要注意清理遗留：例如当前DestroyProjectile里PhotonNetwork.Destroy已经销毁对象 [29] , 重构时可以绕过Photon的销毁，用对象池持久存在，这可能需要PhotonView的Reuse技术或自定义Photon对象池。PhotonPun有简易对象池接口，可实现 IPunPrefabPool 来接管 Instantiate/Destroy，重构可以考虑使用，使PhotonNetwork.Instantiate 自动从池获取对象。

- 对象池的引入，不仅减少GC和Instantiate开销，而且**简化网络逻辑**：当使用自定义池后，PhotonNetwork.Instantiate 调用可被管理器统一封装或替代。例如ProjectileManager可以预先创建带PhotonView的对象，并手动分配ViewID给远端同步，而不是频繁通过Photon服务器创建销毁。此部分实现相对复杂，但收益是网络流量和延迟的降低。

- **工厂封装网络细节**: 将此前分散在各处的 Photon 调用集中到工厂：

- ProjectileManager.CreateProjectile 内部判断：本地是权威玩家（photonView.IsMine）则创建/启用对象，并**发送RPC**通知其他客户端。RPC内容包括：投射物类型标识、初始位置、方向、速度、Damage等信息。远端收到后，调用 ProjectileManager.OnRemoteProjectile 来本地生成一个对应的**视觉子弹**。视觉子弹可以是相同的预制但带不同逻辑（如不检测碰撞，只播放Trail然后在预计寿命后自动回收），也可以是更轻量的表现对象（如仅粒子效果）。
- 由于**只由权威端模拟物理并处理碰撞**，远端的视觉子弹不参与物理，不会重复调用Damage。这避免了网络上多次伤害。DamageSystem应配合这个策略：只有权威端真的调用ApplyDamage并通过StatusManager或其他途径同步减血结果给目标玩家（比如调用目标PlayerStatusManager的RPC）。这样保证一次命中只扣一次血。

- 如果使用Photon的对象同步而不走RPC，也可以：本地权威生成Projectile对象（PhotonNetwork.Instantiate或PhotonView.AllocateViewID），其他客户端会自动创建实例，全部运行ProjectileBase。但为防止多端重复伤害，可以利用photonView.IsMine判断只有owner执行Damage，其它客户端将Projectile标记为“ghost”仅用于显示。当前ProjectileBase.ShouldIgnoreCollision已经排除了_sourcePlayer自身 [44]，可扩展为如果!photonView.IsMine则不调用ProcessHit，只走NetworkHit事件播放效果 [45]。但这方案复杂度高于RPC广播，重构可优先考虑RPC+本地模拟的做法。

- **缓存与性能**: ProjectileManager可以在后台定期清理长时间未用的对象池（或根据场景切换回收），以平衡内存。当前代码已有 CleanupRoutine 协程设想 [46]，可继续完善，例如每隔_cleanupInterval检查池大小和活动数来销毁多余对象。

- **工厂与Weapon交互示例**:

```
// WeaponBase/ProjectileWeapon.FireImplementation
bool FireImplementation(Vector3 dir) {
  ProjectileManager.Instance.SpawnProjectile(this, dir);
  return true;
}
// ProjectileManager
public void SpawnProjectile(WeaponBase weapon, Vector3 direction) {
  // 计算Spawn位置
  Vector3 position = weapon.GetMuzzlePosition();
  var config = weapon.WeaponData.ProjectileConfig;
  // 从池获取Projectile对象
  ProjectileBase proj = GetPooledProjectile(config.prefabName);
  proj.Configure(config, weapon);
  proj.transform.Set(position, Quaternion.LookRotation(direction));
  proj.Launch(direction, config.Speed, weapon, weapon.transform.root.gameObject);
  if(photonView.IsMine && weapon.WeaponData.SyncProjectiles) {
    // 发送RPC创建远端视觉弹道
    photonView.RPC("RpcSpawnProjectile", RpcTarget.Others,
weapon.WeaponData.ProjectileTypeId, position, direction);
```

```
    }
}
```

以上伪码体现了工厂的职责：从池获取对象、配置参数、启动投射物，并在需要时同步远端。而Weapon仅调用一次 SpawnProjectile，不涉及对象如何产生。

5. 网络同步逻辑重构

集中网络同步于管理器，确保所有玩家对武器状态的共识：

- **统一射击同步**: 如前述，通过 PlayerStatusManager 来统一处理射击和切换广播：
- 本地玩家开火成功后，StatusManager调用 `photonView.RPC("RpcPlayerFired", Others, weaponIndex, direction, timestamp)` 将武器序号、方向、时间发送。
- 远端收到 RpcPlayerFired，在 RpcPlayerFired 实现中，通过 `playerWeaponInventory.SwitchToWeapon(index)` 确保远端玩家切换到正确武器（如果尚未切换），然后调用远端的 `currentWeapon.NetworkFire(direction, timestamp)` 方法。对于ProjectileWeapon，NetworkFire 已实现为播放枪口火光+声音并创建视觉弹道 [33]；对于HitscanWeapon，可实现NetworkFire为在准星处闪烁击发效果等。NetworkFire 执行过程中**不产生真正伤害**，仅作表现。
- 此时权威端的Projectile已经在飞并且会调用DamageSystem，DamageSystem再通过例如PlayerStatusManager.UpdateHealth RPC通知受击玩家减血。这保证了**伤害只结算一次**且由权威端决定。

- 如果使用Photon的对象同步而非RPC，也需遵循这个单点伤害原则，可以在DamageSystem中添加判断“如果不是MasterClient/owner则不实际扣血”。

- **统一武器切换同步**: 类似地，当玩家切换武器时，PlayerStatusManager通过RPC `RpcSwitchWeapon(newIndex)` 通知其他客户端调用对应远端PlayerWeaponInventory.SwitchToWeapon。同样WeaponBase.OnWeaponEquipped事件也可以用于在本地更新动画和UI，远端由RPC驱动动画/UI。

- 由于当前 PlayerWeaponController 已有 OnRemoteWeaponSwitch RPC（弃用） [47]，重构后可以删除这些弃用代码，转为在PlayerStatusManager统一实现新的RPC逻辑，减少歧义。

- **移除多余PhotonView**: 如果引入中心管理RPC，对于每把枪、每颗子弹未必都需要各自PhotonView组件。例如WeaponBase目前继承MonoBehaviourPun，大概是为了用PhotonView.IsMine检查本地武器。实际上武器作为玩家子对象，可以通过玩家PhotonView识别owner，不一定要每武器都有ViewID。重构可以考虑**取消武器上PhotonView组件**（ValidateConfiguration里有检查要求Prefab有PhotonView [48]，这可以在重构中移除），以减少Photon网络对象数量。

- 取而代之，用玩家的PhotonView和Weapon索引来标识唯一武器。比如网络消息传 weaponIndex=1 即表示玩家当前使用第2把武器，无需单独PhotonView。

- 子弹对象如果采用“不网络Instantiate只远端模拟”的策略，也可以不需要PhotonView组件（完全本地对象）。只有ProjectileManager需要一个PhotonView用于RPC通信。因此，可显著减少网络同步开销和复杂度。

- **确保一致与安全**: 网络重构需注意边界情况处理：

- 延迟补偿：ProjectileWeapon.NetworkFire中已有基于PhotonNetwork.Time的延迟计算 [49] , 在RPC广播射击事件时可以附带时间戳并在远端CreateVisualTrail时考虑这个 timeDiff，让远端的弹道起始位置向前模拟timeDiff秒的位置，从而提高命中特效同步准确度。
- 状态校验：如玩家快速切枪或断线，应在StatusManager中增加保护，例如不处理过期或非法的 RPC指令（weaponIndex超出范围等），以免引发错误。
- **兼容性**: 重构期间可临时保持一些旧逻辑以平滑过渡。例如保留 OnRemoteFire RPC 一段时间但打日志警告，引导尽快切换到新StatusManager机制。待测试稳定后，再删去旧逻辑。

6. 扩展性的支持

通过上述抽象和解耦，新结构对未来新武器类型将更加友好：

- **新增武器类型**只需：
- 创建对应的 WeaponBase 子类，实现 FireImplementation 和必要的Reload/Equip覆盖。
- 在 WeaponData 或配置资产中加入该武器的参数（若有特殊参数可扩展 WeaponData 或使用继承的 ScriptableObject）。
- 在UI/动画配置中添加新武器的图标和动画状态（如果共用动画参数则可能不需额外改动）。
- 不需要改动核心管理逻辑。由于Inventory和StatusManager基于WeaponBase接口工作，新武器自动融入装备列表和网络同步。比如添加 FlamethrowerWeapon（火焰喷射器，持续射击AOE锥形），可继承 HitscanWeapon 实现持续伤害，每帧消耗弹药并触发DamageSystem喷火伤害，无需修改WeaponInventory或DamageSystem其他部分，只在DamageSystem中扩展DamageType处理即可。

- 投射物类型扩展：如要新增特殊弹药（黏弹、诱导弹），可继承 ProjectileBase并实现独特行为（黏在表面后延时爆炸等），并确保WeaponData的 projectileConfig 有相应字段。ProjectileFactory对新类型不需要特别修改，因为配置会指向新的prefab，工厂照常实例化。若需要统一管理可通过ProjectileBase派生类的Configure来处理特殊参数。

- **减少修改现有代码**：重构方案尽量**保持对现有结构的兼容**：

- WeaponData 尽管内部组织改变（比如嵌入 ProjectileSettings），对策划/设计师使用的界面仍可表现为以往的参数组合，只是更清晰分类。已有武器资产(.asset)可以通过脚本迁移其字段到新结构，避免人工重配。
- WeaponBase 对外接口 (TryFire, TryReload 等) 不变，事件名称不变，UIManager等模块接口不变。因此UIManager、AnimationController基本无需改动逻辑，只是事件触发时机和来源更可靠了。比如之前 OnReloadStarted 事件是在WeaponBase.StartReload触发，现在仍可在开始换弹时触发，UI更新流程不受影响 [17] [18] 。
- Player操作手感方面，也不会因为架构变化受到负面影响——重构提升了同步效率和安全性，玩家几乎察觉不到内部变化，除非专门调整了某些延迟补偿算法使体验更平滑。

综上，新的架构通过接口和管理器明确了模块职责，通过数据驱动和工厂模式减少了硬编码耦合，并为不同武器形态的共存奠定基础。

重构实施步骤清单

为平稳进行重构，可按照以下步骤逐步实施，每步保证现有功能尽量可用，再过渡到下一步：

1. **抽象分类武器类型**：创建 HitscanWeapon、MeleeWeapon 等继承自 WeaponBase，实现基本 Fire 功能（暂时先不集成DamageSystem，直接简易输出调试）。将现有武器逐一归类为 ProjectileWeapon 或 HitscanWeapon 子类，确保游戏运行行为暂时不变。

2. **引入 DamageSystem**：实现一个 DamageSystem 单例和 DamageInfo 结构，迁移原有 IDamageable 直接调用逻辑：

3. 修改 StandardProjectile.ProcessHit 等，将原来的直接 `damageable.TakeDamage(damageInfo)` 替换为 `DamageSystem.ApplyDamage(damageable, damageInfo)`，内部调用 TakeDamage。

4. 检验玩家生命管理仍然正常工作（DamageSystem作为中间层不应改变结果）。此步完成后，子弹伤害都由DamageSystem调度，方便之后添加新伤害类型。

5. **集中网络同步**：重构 PlayerStatusManager 承担武器同步：

6. 实现 RPC 如 RpcPlayerFired, RpcSwitchWeapon 等，在本地调用 WeaponInventory 和远端调用 NetworkFire。
7. 修改 PlayerWeaponController.TryFire：本地开火成功后不再PhotonView.RPC通知，而是交给 StatusManager.OnFireAttempt处理（可以通过监听 WeaponBase.OnWeaponFired 或直接函数调用）。暂时保留原PhotonNetwork.Instantiate子弹方式，以确保射击功能短期内不中断，只是改变通知路径。

8. 测试多客户端下射击和切换的同步是否正常，直至旧的 PlayerWeaponController.OnRemoteFire 确认可弃用，然后删除这些旧RPC方法。

9. **实现 ProjectileManager 与对象池**：

10. 创建 ProjectileManager 脚本，挂载在网络场景常驻对象上，设置PhotonView。实现基本的 Register/Unregister 和 SpawnProjectile 接口，先不启用批量优化，仅实现从 PhotonNetwork.Instantiate 切换为 Photon自定义池：
11. 为 ProjectilePrefab 注册 PhotonPrefabPool，使 PhotonNetwork.Instantiate 实际调用 ProjectileManager 内的创建函数。从而渐进替换武器中 `PhotonNetwork.Instantiate(prefab)` 为调用 ProjectileManager.SpawnProjectile。确保本地与远端都正常生成子弹。
12. 引入简单池机制：维护 Dictionary\<string, Queue\<GameObject>>，使 SpawnProjectile 先尝试队列，有则复用对象，没有才 PhotonNetwork.Instantiate。DestroyProjectile修改为 PhotonNetwork.Destroy改为Queue回收。由于Photon有严格要求，可能需要Hybrid方式：本地物理对象直接回收，远端依然Destroy，但Destroy时判断如果是池对象就仅隐藏不真的销毁。

13. 逐步调优：子弹数量不多时可暂保持PhotonInstantiate，为验证池逻辑可人工将 _maxActiveProjectiles 设很小触发 ForceCleanup 观察效果 50 。

14. **WeaponBase与工厂联动**：修改 ProjectileWeapon.FireImplementation 使用 ProjectileManager：

15. 去除ProjectileWeapon.Fire里直接的 PhotonNetwork.Instantiate调用 27 28 ，改为 `ProjectileManager.Instance.SpawnProjectile(this, spawnPos, direction, finalVelocity)`.
16. 调整 ProjectileBase.Launch：去掉其中对 _damage,_speed 等的设置，因为现在这些应通过 Configure传入。在 SpawnProjectile 时，在Instantiate/复用后立即调用 proj.Configure(config, weapon) 设定damage等，然后 proj.Launch。

17. 校正 BouncyGun.ConfigureProjectile 等特殊配置路径：可暂时保留但实现内部调用 base.ConfigureProjectile + proj.Configure(weaponData参数)，逐步弃用显式 SetBounceParameters。最终目标是 BouncyGun 不再需要特例代码，但为安全可分两步重构，先确保功能，然后精简代码。

18. **精简WeaponBase职责**：整理 WeaponBase和子类代码，应用模板方法模式：

19. 将TryFire整合弹药扣除、冷却标记和事件触发，如上文示例代码所示。检查所有子类的Fire实现，重构为FireImplementation，去掉重复的弹药判断，统一交由基类处理 [13] 。
20. 类似地，整理 TryReload：WeaponBase.TryReload 检查状态后调用 StartReload并触发事件，具体动画（如果有）由AnimationController监听事件。子类基本不需要override TryReload，除非有特殊reload机制，可通过覆写 CompleteReload实现。

21. 清理 WeaponBase 中未使用或重复的内容：如 PlayEmptySound 当前未被调用 [13] [51] ，可以考虑在TryFire判定无法射击且弹药为空时调用，以后UI可通过 OnFireAttempt(false)区分究竟是弹夹空还是冷却中。确保这些辅助方法在需要的地方被正确使用或移除冗余。

22. **测试与验证**：经过上述改造，应对系统各方面进行详尽测试：

23. 单人模式下所有武器的开火、换弹、切枪、命中效果是否正常；弹药数和UI更新是否正确同步 [18] [52] 。
24. 多人联网下：不同延迟条件下，射击命中判定是否准确（有无鬼枪或延迟过大偏差）；武器切换在他人视角是否同步；多种武器混用是否有异常（如激光武器是否在远端也能看到光束）。
25. 投射物特殊情况：弹跳子弹在复杂场景是否正确回收或销毁；高频率射击下对象池有无溢出或错乱；断线重连时ProjectileManager的单例和池是否妥善处理旧对象等等。

26. UI和动画：尤其检查动画触发是否同步，例如装弹动画是否在本地和远端都按照事件时序播放，WeaponUI的弹药计数与Reload进度条是否准确更新。

27. **代码清理与优化**：测试稳定后，清理掉临时兼容代码：

28. 移除 PlayerWeaponController 中不再需要的输入更新和过时RPC。
29. 删去WeaponBase/ProjectileBase中已转移到别处的属性和方法（如WeaponBase.ValidateConfiguration关于PhotonView的检查，ProjectileBase.NetworkDestroy等可能改由管理器处理）。
30. 优化日志输出与Debug开关，使调试信息更聚焦。当前大量的Debug.Log应在提交最终版本前用条件包裹或移除，以免影响性能 [53] [54] 。
31. 最终整理文档和注释，注明新架构下各部分职责，方便后续开发者扩展。

通过以上步骤的循序渐进重构，武器系统将从臃肿耦合走向清晰模块化。在保证现有功能的同时，极大降低了维护成本，并为将来的激光武器、近战武器、AOE伤害等实现提供了**坚实且易拓展**的架构基础。

---

[1] [2] [3] [4] [5] [14] [47] PlayerWeaponController.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/43fcc74c60bcdc50a6b7879b2df59bf550d9d848/Assets/Scripts/Weapons/Core/PlayerWeaponController.cs

[6] [36] WeaponData.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/43fcc74c60bcdc50a6b7879b2df59bf550d9d848/Assets/Scripts/Weapons/Core/WeaponData.cs

[7] [10] [19] [21] [23] [24] [29] [30] [37] [43] [44] [45] ProjectileBase.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/43fcc74c60bcdc50a6b7879b2df59bf550d9d848/Assets/Scripts/Weapons/Core/ProjectileBase.cs

[8] [12] [38] BouncyGun.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/43fcc74c60bcdc50a6b7879b2df59bf550d9d848/Assets/Scripts/Weapons/Weapons/BouncyGun.cs

9 22 25 26 BouncyProjectile.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/43fcc74c60bcdc50a6b7879b2df59bf550d9d848/Assets/Scripts/Weapons/Projectiles/BouncyProjectile.cs

11 16 20 39 40 41 42 StandardProjectile.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/43fcc74c60bcdc50a6b7879b2df59bf550d9d848/Assets/Scripts/Weapons/Projectiles/StandardProjectile.cs

13 15 51 53 WeaponBase.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/43fcc74c60bcdc50a6b7879b2df59bf550d9d848/Assets/Scripts/Weapons/Core/WeaponBase.cs

17 18 52 WeaponUIManager.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/43fcc74c60bcdc50a6b7879b2df59bf550d9d848/Assets/Scripts/Weapons/UI/WeaponUIManager.cs

27 28 33 48 49 54 ProjectileWeapon.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/43fcc74c60bcdc50a6b7879b2df59bf550d9d848/Assets/Scripts/Weapons/Projectiles/ProjectileWeapon.cs

31 32 46 50 ProjectileManager.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/43fcc74c60bcdc50a6b7879b2df59bf550d9d848/Assets/Scripts/Weapons/Network/ProjectileManager.cs

34 35 PlayerAnimationController.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/43fcc74c60bcdc50a6b7879b2df59bf550d9d848/Assets/Scripts/Core/Character/PlayerAnimationController.cs