ChatGPT

模块化武器系统设计方案

1. 现有项目架构对武器系统设计的影响分析

输入系统与事件机制：现有项目采用 Unity 新输入系统并通过事件驱动模式处理输入 [1]。`InputManager` 中定义了例如 `OnFirePressed` 这样的静态事件来代表开火按键 [2]，`PlayerInput` 组件会订阅这些事件并更新自身状态 [3]。这意味着我们的武器系统可以方便地利用这一机制：当检测到玩家触发开火事件时，让当前武器执行射击逻辑。无需直接轮询输入，而是响应事件，符合项目解耦的设计风格。

模块化组件架构：项目将角色控制功能分解为独立组件（如 `PlayerInput`, `PlayerMotor`, `CameraRig` 等），各自负责单一功能，并通过状态管理器或事件协调 [1]。因此，武器系统应设计为**独立的模块**，例如在玩家对象下增加一个**武器控制器**或武器组件，不与移动、重力等耦合。这样可以在不改动核心运动/重力代码的情况下添加武器功能，保持架构清晰。Phase1 文档中明确提到下一阶段将引入武器系统（包含 `WeaponBase`, `Projectile`, `Explosion` 类） [4]，这暗示我们应创建一个武器基类供各种武器继承，实现类似于重力系统中 `GravitySource` 抽象类的结构。

数据驱动配置：项目大量使用 ScriptableObject 实现数据驱动设计，例如移动参数配置使用了 `MovementTuningSO` 脚本对象，方便设计人员调整参数 [1]。为保持一致，我们的武器系统也应采用 **ScriptableObject 配置**武器属性。通过定义武器配置对象，武器的伤害、射速、弹药等都可在 Inspector 中调整，而无需硬编码。这提高了拓展性：添加新武器时，可通过创建新的配置资产来定义其行为。Sprint 功能实现文档也强调了遵循数据驱动设计的重要性 [1]。

网络同步要求：项目使用 Photon PUN2 进行联机。现有网络架构通过在玩家对象上附加 PhotonView 并实现自定义同步组件，实现位置、重力等状态的网络同步 [5] [6]。因此武器系统需要提供**基础的网络同步**支持，例如：各客户端统一玩家持有的武器类型、玩家射击动作的触发以及子弹/投掷物在网络中的生成和移动。可以参考项目提供的示例：例如 PuppetMaster 模块中的 `PUNBallShooter` 脚本使用 `PhotonNetwork.Instantiate` 来在所有客户端生成同步的射弹 [7]。这提示我们，当本地玩家开火时，应在主客户端生成子弹并通过 Photon 实例化广播给其他客户端，保证所有人都能看到相同的弹道。项目计划文档中也提到会新增 `NetworkWeaponController` 组件，通过 RPC 同步射击事件 [6]，这对于**即时命中型**武器（如光束枪）很有效。但对于有实体飞行物的武器（如香蕉或弹力子弹），直接网络实例化实体更直观。总之，我们需在设计中考虑 PhotonView、RPC 调用或网络实例化机制，以实现**武器切换和开火的同步**。

非常规武器需求：本游戏属于休闲派对风格，武器类型怪诞多样，例如“香蕉枪”“黑洞枪”“弹力枪”。这些武器具有特殊效果：香蕉枪可能发射香蕉模型的抛射物，黑洞枪需要生成带引力场的投掷物，弹力枪的子弹可反弹。因此武器系统必须足够灵活，能够通过不同子类或配置支持**多样的射击逻辑**，而不是仅限于传统 hitscan 射线检测。比如黑洞枪可利用现有的 `GravitySphere` 重力源组件来模拟引力吸引 [8]；弹力子弹需要支持物理弹跳。这些需求会影响我们的设计：我们应允许武器通过组合不同的**弹药/投射物组件**实现特殊效果，避免将所有逻辑糅合在一个类中，从而保持结构清晰。

综上，现有架构提供了良好的事件驱动输入和模块化基础，我们将在其之上构建一个易于扩展的武器系统，遵循**抽象基类+数据驱动配置+事件驱动触发+PUN网络同步**的设计原则。

2. 系统结构与类设计说明

为了满足模块化和可扩展性，武器系统将采用面向对象的层次结构和组件化设计。核心思路是定义**武器基类**和**投射物基类**，利用继承和多态实现不同武器的变体；同时通过组合不同组件来增加特殊功能。下图描述了主要类结构：

```
WeaponBase (抽象基类，继承自 MonoBehaviour)
 ├──── ProjectileWeapon : WeaponBase   (典型的发射投射物的武器实现)
 │      ├──── BananaGunWeapon : ProjectileWeapon    (香蕉枪，实现特殊外观或效果)
 │      ├──── BlackHoleGunWeapon : ProjectileWeapon (黑洞枪，可能覆写发射逻辑加入引力场)
 │      └──── BouncyGunWeapon : ProjectileWeapon    (弹力枪，可能覆写发射逻辑支持弹跳特性)
 └──── HitscanWeapon : WeaponBase (可选扩展：即时命中类武器，例如激光枪，使用射线检测)

ProjectileBase (抽象投射物基类，继承自 MonoBehaviour)
 ├──── StandardProjectile : ProjectileBase   (普通子弹/香蕉投射物)
 ├──── BlackHoleProjectile : ProjectileBase  (黑洞投射物，携带 GravitySphere 组件)
 └──── BouncyProjectile : ProjectileBase     (弹力投射物，可弹跳若干次)
```

WeaponBase 类：抽象武器基类，负责定义所有武器共有的属性和行为。该类可以挂载在玩家的武器槽对象上或直接在玩家对象下。主要包含：

- **属性**：引用武器配置数据（ScriptableObject）、开火冷却时间、弹药计数、武器模型等。
- **方法**：
- `Fire()`：武器开火的接口方法。基类实现中处理通用流程，如检测冷却是否结束、扣除弹药等，然后调用抽象的`LaunchProjectile()`或`PerformShoot()`。
- `LaunchProjectile()`：抽象方法（或虚方法），由子类实现具体的射击方式。如果是发射实体弹丸的武器，可能实现为实例化一个投射物Prefab并赋予初速度；如果是射线武器，则实现为射线检测伤害。
- `OnEquip()`/`OnUnequip()`：当武器被玩家装备或收起时的回调，可用于切换模型的可见性或初始化状态。

通过使用抽象基类，能够将不同武器的共同部分（如触发输入、同步逻辑等）集中在一起，实现**代码复用**和**统一接口**。例如，我们可以将武器挂载到玩家对象下，通过`playerWeapon.CurrentWeapon.Fire()`来触发射击，而不关心当前是哪个具体武器。

ProjectileWeapon 类：这是`WeaponBase`的一个具体子类，实现了**抛射物类武器**的通用行为。多数休闲武器（香蕉枪、黑洞枪、弹力枪）都属于发射某种物体的类型，因此可以用这个类统一处理。例如： - 在`Fire()`中，计算射击起点位置和方向（通常从枪口位置，或摄像机朝向），然后实例化对应的弹丸Prefab（从武器配置中取得）。 - 可在`Fire()`内调用基类提供的辅助函数，如处理后坐力动画、播放开火音效等。 - 如果需要考虑持续开火（自动武器），可以在Update中检测`FireHeld`来周期性触发`Fire()`，并由配置的射速限制频率。

对于一些特殊投射物武器，可能直接使用`ProjectileWeapon`不需要进一步子类化，通过配置即可满足需求；而另一些可能需要轻微定制： - **BananaGunWeapon**：可能直接使用`ProjectileWeapon`的实现即可（除非有额外效果），射出的Prefab为香蕉模型的普通投射物。 - **BlackHoleGunWeapon**：可以继承`ProjectileWeapon`，重写`Fire()`或`LaunchProjectile()`：在生成弹丸后，可能设置弹丸的某些参数，例如引力强度或寿命；或者在此武器开火时附加额外效果（比如屏幕抖动等）。如果黑洞弹需要 charging 或特殊触发，也可在此加入。 - **BouncyGunWeapon**：也可继承`ProjectileWeapon`以调整弹丸属性，如设定弹丸的反弹次数上限等。如果基础`ProjectileWeapon`已支持通过配置开启弹跳（例如配置弹跳次数>0则开启弹跳逻辑），则可能无需额外子类。

这样设计使得**添加新武器**非常方便：对于大多数新武器，只需创建新的 ScriptableObject 配置，赋予新的弹丸Prefab和参数，即可使用已有的 `ProjectileWeapon` 脚本实现射击。如果遇到截然不同的射击机制（例如激光束、喷火器），则新建一个继承自 `WeaponBase` 的类覆盖 `Fire()` 行为。

ProjectileBase 类：抽象投射物基类，代表被射出的**子弹/弹丸/投掷物**。每种投射物Prefab都应附加一个继承自此基类的脚本，用于控制其飞行和命中效果。关键职责包括： - **运动方式**：可以通过刚体物理推动（Rigidbody + 初速度）或在Update中按设定速度移动。考虑到项目有自定义重力系统，投射物如果有刚体，应当受到重力场影响，以实现如香蕉在行星引力下抛物线下坠的效果。 - **碰撞检测**：在 `OnCollisionEnter` 或 `OnTriggerEnter` 中检测命中。当投射物碰到玩家或环境时，执行相应逻辑，如造成伤害、生成效果然后销毁自身。 - **生命周期**：可以有最大生存时间（防止无限飞行），倒计时结束后自动销毁。特殊弹丸可能在超时后引发特殊效果（如黑洞弹丸飞一段时间后在空中生成引力场）。

具体的 Projectile 派生类示例： - **StandardProjectile**：标准投射物，如香蕉子弹。附加刚体，在 `Start()` 时根据武器传来的速度参数施加初速度飞出。命中时直接销毁自身并通知命中的对象（例如通过接口或发送消息造成伤害）。 - **BlackHoleProjectile**：黑洞投射物。在飞行过程中或命中时生成**引力场**效果。实现上，可直接在该Prefab下挂一个 `GravitySphere` 组件（配置合适的引力强度和半径） [8] 。当黑洞弹生成后，短暂延迟激活 `GravitySphere`，使其在一定范围产生引力，将周围物体拉拽。引力场可持续数秒后消失（可通过定时销毁 GravitySphere 或将其enabled设为false）。如果引力应该从撞击点开始，则可在 OnCollision 时将弹丸停留在命中位置并开启引力场。同时可以附带粒子特效模拟黑洞视觉。黑洞弹可能对命中的玩家不直接造成伤害，而是通过引力效果间接影响游戏局面。 - **BouncyProjectile**：弹力投射物。其特点是在撞到环境时不立即销毁，而是**弹跳**。实现方式：可以赋予弹丸一个物理材质(Physic Material)设置高弹性，使其物理碰撞自动反弹；同时脚本中维护一个**剩余弹跳次数**计数，每次发生碰撞时减一，减到0时再销毁弹丸。也可以不依赖物理材质，而在脚本的 `OnCollisionEnter` 中手动计算反射方向：例如 `Vector3 reflectDir = Vector3.Reflect(currentVelocity, collisionNormal)`，然后赋予刚体新的速度。这样可以更精确地控制弹跳衰减和次数。弹力弹丸命中玩家时依然要触发伤害，然后销毁或继续反弹（可根据设计决定，通常命中玩家就消耗掉弹丸）。

玩家武器管理：在玩家角色对象上，可以挂一个 `PlayerWeaponController` 脚本（或直接集成在 `PlayerStateManager`）用于管理当前持有的武器。其职责： - 保存当前武器（`currentWeapon` 引用指向一个 `WeaponBase` 对象），以及可选的武器列表用于切换。 - 在 `Update` 中读取 `PlayerInput` 提供的输入状态，触发武器动作。例如：

```
if(playerInput.FirePressed) {
  currentWeapon.Fire();
}
if(playerInput.FireHeld) {
  currentWeapon.HandleHoldTrigger();
}
```

这样，当玩家按下开火键时，调用当前武器的 `Fire()` 方法。对于持续射击武器，`HandleHoldTrigger()` 可处理长按连续发射逻辑（或者也可在 `WeaponBase.Update` 内部处理，取决于实现）。 - 管理武器切换：响应玩家换枪输入（如果有），切换 `currentWeapon` 指向，并启用/禁用相应武器模型和脚本。可以调用各武器的 `OnEquip/OnUnequip` 做善后，例如调整输入过滤（不允许同时使用多个武器输入）。

通过这种设计，武器作为独立组件彼此解耦，`PlayerWeaponController` 充当协调者，确保任一时刻只有当前武器响应开火输入。这防止多个武器组件同时监听输入事件造成冲突。同时，新武器组件只需符合 `WeaponBase` 接口，就能被管理器纳入，无需改动现有代码，体现了**开闭原则**。

类之间的关系： - 每个具体武器类（如 BananaGunWeapon）可以在编辑器中作为预制体存在，包含模型、粒子和其脚本，引用其配置对象。玩家初始携带的武器可以在场景中作为子物体放在玩家下，或者在运行时由 `PlayerWeaponController` Instantiate。简化起见，也可在玩家预制体下预置所有可能武器的子物体，一开始只激活默认武器，其余禁用，需要时切换（这样模型切换和网络同步比较容易处理）。 - 武器配置（ScriptableObject）与武器脚本通过序列化字段关联，武器脚本在Awake/Start时从配置读取参数。投射物Prefab也会在配置中指定，供武器脚本实例化。 - 投射物与武器：当武器实例化弹丸Prefab后，通常会获取该弹丸上的 `ProjectileBase` 脚本，调用其初始化方法（例如传入射击方向、速度向量、伤害值以及开火者引用等）。投射物在生命周期内若需要与武器通信（例如命中后通知武器），可通过这种引用或事件处理。一般来说，大部分逻辑在弹丸自身完成即可，武器不需干预。

此结构实现了**清晰的分层**：玩家输入 -> 武器 -> 弹丸，各司其职。同时利用继承和组件组合，可以轻松添加新类型武器或特殊效果，而不会影响其他部分。

3. ScriptableObject 配置示例及使用

为支持武器参数的灵活配置，我们定义一个 **武器配置 ScriptableObject** 类，例如 `WeaponData`。通过该配置资产，可以在Unity Inspector中调整武器的各种属性，而无需修改脚本代码。这符合项目一贯的数据驱动思路 [1]。示例定义如下：

```
[CreateAssetMenu(fileName = "NewWeaponData", menuName = "GravityShoot/Weapon Data", order = 1)]
public class WeaponData : ScriptableObject {
  public string weaponName;            // 武器名称
  public GameObject projectilePrefab;     // 弹丸预制体
  public float fireRate = 1f;           // 射速（每秒发射次数）
  public int magazineSize = 0;          // 弹夹容量（0表示无限弹药）
  public float reloadTime = 1.5f;        // 换弹时间
  public bool automatic = false;         // 是否长按连续射击
  public float projectileSpeed = 20f;     // 弹丸初速度
  public float damage = 20f;           // 基础伤害
  public int maxBounceCount = 0;         // 最大弹跳次数（针对弹力枪）
  public float gravityForce = 0f;        // 引力强度（针对黑洞枪, 为0则无特殊引力）
  // ... 其他可能的配置，例如爆炸半径、特效Prefab引用等
}
```

- **weaponName**：用于识别武器，可以用于UI显示或调试。
- **projectilePrefab**：此武器发射的弹丸预制件。不同武器可指定不同的Prefab，例如香蕉枪指定香蕉模型Prefab，黑洞枪指定带GravitySphere的弹丸Prefab。
- **fireRate**：射速控制武器的冷却。WeaponBase可以根据 fireRate 计算每次射击后的冷却时间间隔（例如 cooldown = 1/fireRate 秒）。
- **magazineSize & reloadTime**：如果需要弹药机制，可用这两个参数控制弹夹容量和换弹延迟。休闲游戏中或许可以忽略弹药限制，设为无限。
- **automatic**：标记武器是否支持长按连射。如果true，则在玩家长按开火键时每隔一定间隔自动开火；false则每次按下只射击一发。
- **projectileSpeed**：弹丸初速度大小。武器在实例化弹丸后，会根据朝向和此速度设置弹丸的刚体速度。不同武器的弹丸速度可不同（如香蕉可能较慢抛物线，弹力子弹可能高速直线）。
- **damage**：基础伤害值。弹丸命中玩家时将根据此值扣减对方血量（具体伤害计算可能还考虑命中部位或衰减等，这里简化为常数）。

- **maxBounceCount**：弹丸最大弹跳次数。对于弹力枪配置一个>=1的值，其他武器可设为0表示弹道不发生弹跳。弹丸脚本会读取这个值决定是否开启弹跳逻辑。
- **gravityForce**：引力强度。针对黑洞武器的特殊参数，表示黑洞弹丸施加的引力大小。例如配置一个较大的值来显著拉动附近的玩家或物体。该值可用于在生成 GravitySphere 时设定其 Gravity 属性。如果武器没有特殊引力效果，留0即可。

（注：以上只是示例字段集，实际实现可根据需要增减参数，并可以细分为多种ScriptableObject。如果某些武器差异极大，也可以为它们定制独立的配置类，继承自通用WeaponData。）

使用方式：在 Unity 编辑器中，开发者可以通过右键菜单「Create > GravityShoot > Weapon Data」创建新的武器配置资产 [9] 。创建后，在 Inspector 中填入该武器的各项属性。例如：

- BananaGunData: `weaponName = "Banana Gun"`, 指定香蕉模型的 projectilePrefab（假设已制作一个BananaProjectile预制体带香蕉模型和StandardProjectile脚本）, `fireRate = 1`, `automatic = false`, `projectileSpeed = 25`, `damage = 10`, `maxBounceCount = 0`, `gravityForce = 0`。
- BlackHoleGunData: `weaponName = "Black Hole Gun"`, projectilePrefab 指向 BlackHoleProjectile 预制体（内含 GravitySphere 组件）, `fireRate = 0.5`, `automatic = false`, `projectileSpeed = 15`, `damage = 0`（黑洞可能不直接伤害）, `maxBounceCount = 0`, `gravityForce = 9.8`（例如模拟重力吸引强度，可根据体验调整）。
- BouncyGunData: `weaponName = "Bouncy Gun"`, projectilePrefab 指向 BouncyProjectile 预制体, `fireRate = 1`, `automatic = true`（假设弹力枪可连射）, `projectileSpeed = 20`, `damage = 15`, `maxBounceCount = 3`（弹跳3次后消失）, `gravityForce = 0`。

在场景中，将武器组件挂载到玩家对象（或其子对象）上，并关联相应的WeaponData配置。例如给玩家添加 `BananaGunWeapon` 脚本组件，然后在Inspector中将其 `config` 字段设置为 BananaGunData。这样，该脚本会根据配置初始化武器参数。开火时，它会读取config里的 projectilePrefab 来生成香蕉子弹。若实现了武器切换系统，也可以在运行时通过代码赋值或预先挂载多个武器组件，每个绑定不同的config。

通过ScriptableObject配置，**新增一种武器**的流程将非常简洁：美术创建模型和弹丸Prefab，程序创建新WeaponData资产配置属性，然后在玩家上添加对应的武器脚本引用此配置即可。绝大部分武器行为差异都能通过配置数值或Prefab来体现，保证了系统的**灵活性**和**可扩展性**。

4. 示例武器实现说明

下面结合上述设计，分别简要说明三种示例武器在系统中的实现方式和特殊逻辑：

香蕉枪 (Banana Gun)

香蕉枪属于**发射实体弹丸**的武器，但它的趣味在于弹丸外观为香蕉模型。实现上，香蕉枪可以直接使用 `ProjectileWeapon` 基类提供的默认开火行为，无需特殊子类。其配置（`BananaGunData`）指定 **BananaProjectile** 预制体，该预制体附带一个标准的弹丸脚本。例如 `BananaProjectile` 脚本可以直接继承自 `ProjectileBase`，内部并无特别之处：在 Start 时获取 Rigidbody，按照 `projectileSpeed` 赋予初速度；碰撞时产生粒子效果（碎香蕉之类的特效）然后销毁即可。香蕉枪的伤害可以设定中等偏低（例如10点），没有额外效果或弹跳。由于香蕉形状可能不太稳定，我们可以调整其碰撞体积使命中判定合理。总之，香蕉枪是**最基础**的武器，实现难度低：模型美术上与普通枪不同，但代码层面与传统枪械类似。它验证了我们的武器系统可以轻松支持替换弹丸模型的需求——只需更换Prefab和配置参数，逻辑无需改动。

黑洞枪 (Black Hole Gun)

黑洞枪的独特之处在于发射**带有引力场效果**的投掷物。玩家开火后，弹丸飞出并在一定条件下产生“黑洞”吸引周围物体。我们通过结合**GravitySphere**组件来实现这一效果[8]。具体方案： - 黑洞枪继承自`ProjectileWeapon`。在其配置(`BlackHoleGunData`)中，指定的 projectilePrefab 为 **BlackHoleProjectile**。这个Prefab上挂有`BlackHoleProjectile`脚本，且包含一个`GravitySphere`子组件或在运行时动态添加。 - 当黑洞枪开火时，`ProjectileWeapon.Fire()`实例化 BlackHoleProjectile。BlackHoleProjectile 脚本会在启动时暂时**关闭** **GravitySphere**（避免刚发射就把周围玩家拉偏），并给自身一个向前的初速度。 - BlackHoleProjectile 可以设置一个**引爆逻辑**：例如在命中环境或飞行了固定时间后，进入“爆炸”阶段——此时将自身速度设为0（停留在引爆点），激活 GravitySphere 组件并设置其重力强度和半径（可从武器配置的 gravityForce 读取）。这样，一个持续数秒的引力场黑洞就出现在场景中，附近的玩家或物体的自定义重力将受到该 GravitySphere 影响，被拉向弹丸位置[10]。引力公式由 GravitySphere 基类提供，支持半径内随距离衰减的吸引力[11]。 - 在引力场作用期间，可以播放黑洞特效（旋涡状粒子等）加强视觉反馈。数秒后，引力场消失（可销毁整个弹丸对象或仅移除 GravitySphere）。 - 黑洞枪的伤害：视设计而定，可能**不直接造成伤害**，主要用途是控制场面、推拉对手。如果需要伤害，可在黑洞消失时对范围内玩家造成一次性伤害或者完全依赖于被摔落等间接伤害。 - 由于黑洞效果可能影响游戏平衡，我们通过 ScriptableObject 配置轻松调节其强度和持续时间，不需改代码。比如调高 gravityForce 可以增加拉力，增加 GravitySphere.radius 可以扩大吸引范围等。

通过上述实现，黑洞枪展示了武器系统对**复杂效果**的支持：我们充分利用了项目现有的 GravitySphere 模块，无需重新编写引力计算逻辑，只需将其作为武器效果的一部分配置进去。这种设计也体现了组件组合的威力——武器发射出一个带特殊组件的弹丸，从而实现丰富的功能。

弹力枪 (Bouncy Gun)

弹力枪发射的子弹不会像普通子弹那样命中即停，而是会在碰到墙壁地形时**弹跳**数次，从而有机会拐弯撞击躲在掩体后的玩家，增加了游戏的趣味和不可预知性。它的实现细节如下： - 弹力枪可以使用`ProjectileWeapon`的逻辑发射弹丸，但我们需要一个专门的弹丸类型 **BouncyProjectile** 来处理碰撞反弹。BouncyProjectile 脚本继承自`ProjectileBase`。 - 在 BouncyProjectile 中，我们可以预设一个物理材质(PhysicMaterial)给予Collider，使其弹性接近1。这将让Unity物理引擎在碰撞时自动计算反射速度向量，实现弹跳效果。同时在脚本中维护一个`bouncesRemaining`计数，初始值从武器配置的`maxBounceCount`读取（例如配置为3则可弹3次）。每次`OnCollisionEnter`时，`bouncesRemaining--`。当减为0时，表示弹丸弹跳次数已用完，下一次碰撞就销毁弹丸。 - 如果不用物理材质，也可在脚本中自行实现：每次碰撞时获取碰撞法线 normal 和当前速度 velocity，计算`Vector3 reflected = Vector3.Reflect(velocity, normal)`，然后设刚体速度为 reflected * 某系数（如0.8保持部分能量，或1保持全速）。同时减少计数。这样可更精细地控制每次弹跳速度衰减。 - 弹力子弹在反弹过程中依然可以伤害玩家：当`OnCollisionEnter`检测到碰撞对象是玩家且尚有伤害未造成过，则对该玩家造成伤害（damage值可从配置传入）。为了避免一次子弹多次伤人，可以设置弹丸在首次撞击玩家后就销毁，或在玩家上设置触发器碰撞单独处理。 - 弹力枪配置（`BouncyGunData`）会将`maxBounceCount`设为>0（如3），表示启用弹跳逻辑。自动/连续射击 (automatic) 可根据需要设为 true，让玩家长按时连续发射弹力弹（想象霰弹枪跳弹雨的效果）。不过射速可能需要限制以避免场景出现过多弹跳物。 - 需要注意的是，多次弹跳会使子弹存留时间变长，我们应设置一个**最大生存时间**（比如3秒），到时强制销毁，防止无限弹跳占用资源或影响游戏节奏。这可在Projectile脚本的Update中倒计时实现。

弹力枪验证了我们设计的弹丸机制可以扩展出弹跳这样略复杂的物理行为，而且主要逻辑被封装在弹丸自身，不需要修改武器开火流程。设计人员可以通过调整弹跳次数、弹力系数等配置改变武器表现，保持了系统的**拓展性**和**可调节性**。

其他可能的武器扩展

（可选说明：我们的系统还能支持更多奇思妙想的武器，例如**散弹枪**（一次发射多个弹丸——可在Weapon子类的Fire中生成多个Projectile；配置可增加一个弹片数量参数）、**激光枪**（HitscanWeapon子类，用射线立即命

中目标；可参考项目NetworkWeaponController用RPC广播射线结果的方案[6]）、**粘弹枪**（弹丸黏在物体上延迟爆炸；可在Projectile脚本用Collision判断黏附并在延时后触发Explosion效果）等等。这些都可以在当前架构下通过新增派生类和配置来实现，证明了设计的灵活性。)*

5. 网络同步的基础支持方案

在多人游戏中，为保证所有客户端对武器行为的统一感知，我们需要对武器的关键状态和动作进行网络同步。考虑到本项目使用 Photon Pun2，我们将遵循其范式实现**最小可行**的同步功能：

武器状态同步：首先，每个玩家当前持有的武器类型需要同步给其他客户端。简单做法是在玩家的 `PhotonView` 同步流中加入一个武器ID，或者在武器切换时发送RPC通知。例如，给玩家添加一个脚本实现 `IPunObservable`，每帧将 `currentWeaponId` 通过 `PhotonStream.SendNext` 发送；在接收端据此激活相应武器模型[12]。这样，当玩家换枪时，其他人会收到更新，显示正确的武器模型/名称。如果游戏中武器不会频繁切换，此同步也可以改为事件驱动：当本地玩家换枪时调用 `photonView.RPC("OnWeaponChanged", RpcTarget.Others, weaponId)`。

射击动作同步：对于**发射实体弹丸**的武器，我们采用 **PhotonNetwork.Instantiate** 来生成网络对象。具体而言，当本地玩家按下开火键且通过冷却检查后：- 本地调用 `PhotonNetwork.Instantiate(projectilePrefab.name, position, rotation, ...)` 来创建弹丸。这会在所有客户端实例化一个带 `PhotonView` 的弹丸对象[7]。该弹丸预制体应设置在 Photon 的资源中或已注册，否则无法网络实例化。- 生成时可以通过 `Instantiate` 的附加数据参数传递一些初始信息给弹丸对象，例如初速度向量、伤害值等。示例中 `PUNBallShooter` 就传递了velocity数据[13]。我们的弹丸脚本在 `OnPhotonInstantiate` 或 Start 中读取这些数据，赋予自身刚体。同样地，可以传递射手的ID用于伤害归属判断等。 - 由于弹丸具有 PhotonView，默认所有客户端都会各自模拟物理。但是Unity物理计算并非完全确定性，可能出现细微差异。为确保高度一致，可以让**发射者客户端**作为弹丸的Ownership并**主导模拟**：即弹丸对象上加 `PhotonRigidbodyView` 或 `PhotonTransformView` 组件，把刚体位置同步给他人。这样其他客户端会跟随主客户端的弹丸运动状态，避免模拟差异。对于休闲游戏而言，也可以不严格同步每帧位置，只要弹丸飞行时间短差别可忽略。不过出于稳妥，我们建议启用基础的刚体位置同步。- 当弹丸在主客户端发生碰撞击中玩家时，需要通知所有人该碰撞结果。典型做法是在碰撞检测处，如果击中了另一个玩家的碰撞体，调用对方的 `PhotonView.RPC("TakeDamage", RpcTarget.All, damageAmount)`，让所有人（包括被击者自己）执行扣血处理[14]。这样健康值变化在各端保持一致。若弹丸有AOE（范围伤害），也以RPC广播方式处理。- 对于**即时命中类武器**（虽然题目例子没有，但可扩展），可以不生成实体，而是采用RPC广播射击信息的方法。即本地执行射线检测后，通过 `photonView.RPC("FireWeapon", RpcTarget.Others, origin, direction, timestamp)` 通知他人，同步调用开火效果和命中判定[6]。项目的 `NetworkWeaponController` 草案正是如此设计的[15][6]。我们的架构允许未来增加类似的 `HitscanWeapon` 来使用这种同步策略。

动画和特效同步：玩家开火通常伴随动画（如开火手部动画）和特效（枪口火焰、声音）。为了简化，实现**最低限度**的同步，可利用以下策略：- **弹丸可见即可证实射击**：对于发射型武器，其他玩家看到弹丸出现就知道你开火了。因此是否同步开火动画并不影响游戏理解。但为了更好的体验，可以对枪口火焰和开火声音进行同步。 - **Animation/Particles**：若使用角色动画控制枪械开火动作，可在Animator参数上加触发，在开火时本地触发参数并通过 PhotonAnimatorView 自动同步给其他客户端（PhotonAnimatorView 会同步 Animator 状态）。或者更直接，在开火RPC里播放特效：例如在 `FireWeapon` RPC的实现中，各客户端各自调用 `PlayShootEffects(origin, direction)` 来产生枪口闪光、声音[16]。因为我们知道开火起点和方向，也可以在远端重建一些效果。- **武器切换动画**：类似的，也可通过同步当前武器ID后，在客户端触发换武器动画。

网络带宽考虑：基础的武器同步不会产生大量数据——开火事件相对少，并且弹丸生成也只是创建对象而非持续大量数据。PhotonNetwork.Instantiate内部是一次性广播，对带宽影响小。如果有很多弹丸同时存在，开启Transform同步会有一些持续流量，但PUN2允许调节发送率。可以根据需要对不同弹丸调整 PhotonView 的 `SynchronizationRate`。鉴于这是休闲游戏，精度要求没那么严苛，**最小可行方案**下不进行高级优化，如服务

器验证或客户端预测。不过，如果将来需要提升体验，可以参考项目网络方案文档中的做法添加简易**客户端预测**（比如对本地玩家的武器射击即时响应，再通过网络校正）。

总结来说，武器系统的网络同步在本设计中主要包括： - **武器选型同步**：确保各客户端的一致显示（通过PhotonView数据流或RPC实现）。 - **射击动作同步**：通过PhotonNetwork.Instantiate共享弹丸实体；若为非实体武器则通过RPC共享射击结果。 - **命中效果同步**：利用RPC在所有客户端执行伤害和特效处理，保持游戏状态一致。

这样的实现满足了题目要求的**基础网络同步**：玩家看到彼此使用相同的武器，射击时能看到对方射出的香蕉、黑洞和弹跳子弹，并且受到相应影响。这一切都融入在现有PUN2网络框架下，遵循了项目既定的同步模式 [12] [6]。通过上述模块化武器系统设计，我们实现了结构清晰、易扩展且能在联网环境下正常工作的武器框架，可为日后增加更多有趣的武器奠定基础。

来源：现有项目架构与文档 [2] [3] [4] [1]，以及Photon PUN2 使用示例 [7] 和网络同步方案参考 [6]。

---

[1] SPRINT_FEATURE_IMPLEMENTATION.md
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/3c6eec60c31cb18d7b98959318780b63677555e0/Docs/SPRINT_FEATURE_IMPLEMENTATION.md

[2] InputManager.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/3c6eec60c31cb18d7b98959318780b63677555e0/Assets/Scripts/Core/Character/InputManager.cs

[3] PlayerInput.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/3c6eec60c31cb18d7b98959318780b63677555e0/Assets/Scripts/Core/Character/PlayerInput.cs

[4] [12] README_Phase1.md
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/3c6eec60c31cb18d7b98959318780b63677555e0/Docs/README_Phase1.md

[5] [6] [14] [15] [16] PUN2_NETWORK_IMPLEMENTATION_PLAN.md
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/3c6eec60c31cb18d7b98959318780b63677555e0/Docs/PUN2_NETWORK_IMPLEMENTATION_PLAN.md

[7] [13] PUNBallShooter.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/3c6eec60c31cb18d7b98959318780b63677555e0/Assets/PuppetMaster-PUN2/PUNBallShooter.cs

[8] [10] [11] GravitySphere.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/3c6eec60c31cb18d7b98959318780b63677555e0/Assets/Scripts/Core/Gravity/GravitySphere.cs

[9] MovementTuningSO.cs
https://github.com/DR9K69AI79/DMT318_ASGM2_GravityShoot/blob/3c6eec60c31cb18d7b98959318780b63677555e0/Assets/Scripts/SO/MovementTuningSO.cs