

RaffleMachine 详细报告书

项目名称: RaffleMachine

作 者: 唐祥霖

GitHub : <https://github.com/xianglintang>

邮 箱: xianglintang@outlook.com

目录

遇事不决抽奖器详细报告书	3
一、引言	3
二、准备阶段	3
三、开发阶段（主要介绍非淘汰，因为淘汰跟非淘汰大体一致）	4
1. 介绍预制件及附着于预制件的脚本和预制体相关功能详解	4
(1)非淘汰抽奖方面：	4
(2) 淘汰抽奖方面：	6
2. 首先介绍贯穿全项目的全局单例类 GlobalTransferStation	9
(1) NotEliminateItemInformationPanel 和 EliminateItemInformationPanel:	9
(2) TipMessageText:	9
(3) InititalFontSize:	9
(4) NotEliminate_AllOdds 和 Eliminate_AllOdds:	9
(5) 非淘汰方面:	9
(6) 淘 汰 方 面 的 ListEliminateItem 、 ListEliminateItemData 、 CurrentCheckedEliminateItem 与第 5 点非淘汰方面同理。	10
(7) RaffleIsRuning	10
(8) createOrLoadData	10
3. 介绍非淘汰机制的添加修改删除以及抽奖逻辑	11
(1) 添加逻辑（位于 CreateNotEliminateItem 脚本）:	11
(2) 修改逻辑（位于 UINotEliminateItemInformation 脚本）:	11
(3) 删除逻辑（位于 UINotEliminateItemInformation 脚本）:	12
(4) 抽奖逻辑（位于 NotEliminateManagement 脚本）:	12
4. 介绍淘汰机制的添加修改删除逻辑以及抽奖逻辑	12
(1) 添加逻辑（位于 CreateEliminateItem 脚本）:	13
(2) 修改逻辑（位于 UIEliminateItemInformation 脚本）:	13
(3) 删除逻辑（位于 UIEliminateItemInformation 脚本）:	13
(4) 抽奖逻辑（位于 EliminateManagement 脚本）:	13
5. 介绍存档机制（位于 CreateOrLoadData 脚本类）	14
(1) 介绍创建文件已经创建存档路径	14
(2) 介绍写入存档，即保存存档	14
(3) 介绍读取存档	15
(4) 介绍淘汰抽奖管理类怎么在读取存档后顺利恢复选项准确位置和状态	15
四、 未来可继续优化方面	16
1. 非淘汰抽奖的算法优化	16
五、 结尾总结	16
当然缺点也是很多的，缺乏趣味性，确实略感无聊，单独是这样的，如果能跟其他游戏项目 合并，淘汰抽奖添加条件，每次抽完，条件变多，很上头的。	16

遇事不决抽奖器详细报告书

一、引言

本抽奖器为 unity 引擎，使用 C# 语言开发的一款安卓软件，也可以称为游戏。

本抽奖器是我从我原创的文字游戏里面独立出来翻新出来的，添加自定义内容和概率模块，并且提供自带存档功能，即删除，修改，添加操作，自动存档，更好的可玩性。

以下图 1 提供项目最终图片



图 1 面板主要层级和最终面板

二、准备阶段

开发环境：Unity 2022.3.15f1c1

开发语言：C#

开发主要思路流程（思路比较简洁，后面会详细解说）：

1. 首先开发非淘汰抽奖器，建立非淘汰抽奖选项数据类，然后建立预制件，然后建立非淘汰抽奖选项数据类脚本，将脚本挂载预制件上，然后预制件将其内的子级物体拖拽到脚本对应变量完成赋值提前操作。
2. 非淘汰存档机制，前面建立了数据类，其实就是将数据类转为 json 格式，然后写成 json 文件进行保存，此外，修改，删除，添加，都会自动调用存档机制
3. 读取存档，我们读取存档后，根据 json 文件，逆转为相应数据类，然后我们根据一系列规则，去重新自动创建。
4. 抽奖机制，就是每个自定义概率总和，然后每个自定义概率/总和概率，得到各自比例为

最终概率，然后抽奖的时候，我们随机一个数，并且定义一个数 `sum` 为第一个选项的最终概率，然后循环判断存在的选项，如果随机概率不小于选项，则 `sum=sum+选项的最终概率`，继续循环，直到随机概率小于 `sum`，由于 `sum` 是提前的，所以判断的时候，要在 `sum` 变化前；十连抽就是对抽奖机制循环十次，但是抽取信息会搜集起来，抽完再集中显示。其实这种抽奖机制，相当于区间判定。

5. 开发淘汰抽奖器，与上面流程一样，都是独立数据类，独立脚本，独立预制件，但是添加了循环的动画，也就是预制件的本身物体有默认图片，子级有经过扫描的图片，有已经选择的图片，已经选择不同于删除，是可以恢复的，所以抽奖机制就是判断脚本里面的某个变量，是否选择过，选择过了，就跳过，如果没选择，就会进行相关信息处理。

6. 淘汰抽奖的动画，其实也不算动画，是暂停机制，使用协程，对每一次循环暂停一定时间，逐渐减慢，直到最终确认。

7. 存档机制更新到淘汰抽奖，抽奖机制差不多，第一时间就出来了，动画只是动画而已。

三、开发阶段（主要介绍非淘汰，因为淘汰跟非淘汰大体一致）

1. 介绍预制件及附着于预制件的脚本和预制体相关功能详解

(1)非淘汰抽奖方面：

非淘汰抽奖预制件如图 2 所示：

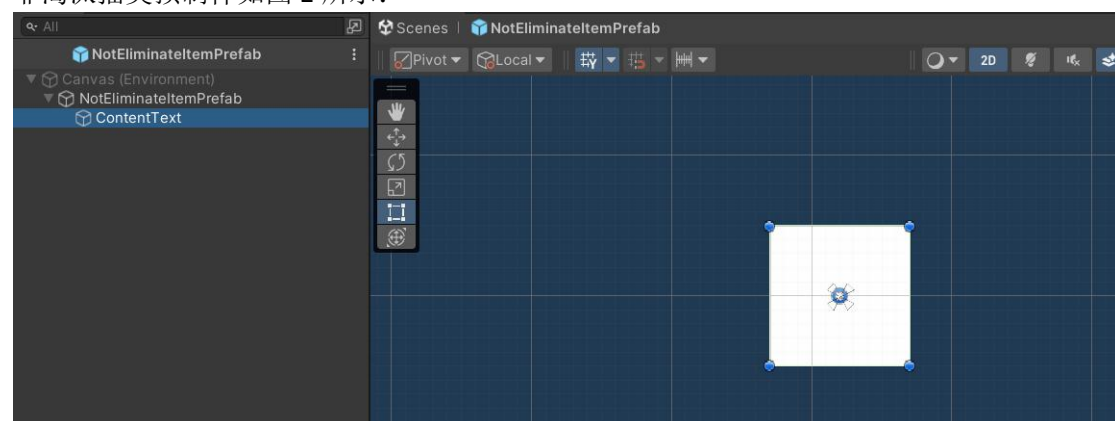


图 2 非淘汰抽奖预制件结构

预制件物体内仅有一个文本物体，来表示内容，物体主要组件如图 3 所示

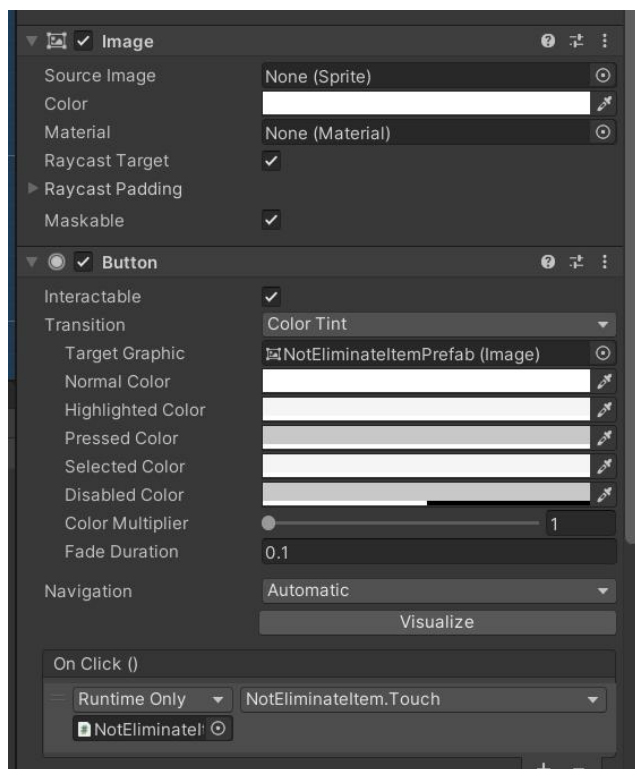


图 3 图片组件和 button 组件

另外相应预制件的脚本为：NotEliminateItem.cs，主要结构如图 4，button 触发代码为图 5。

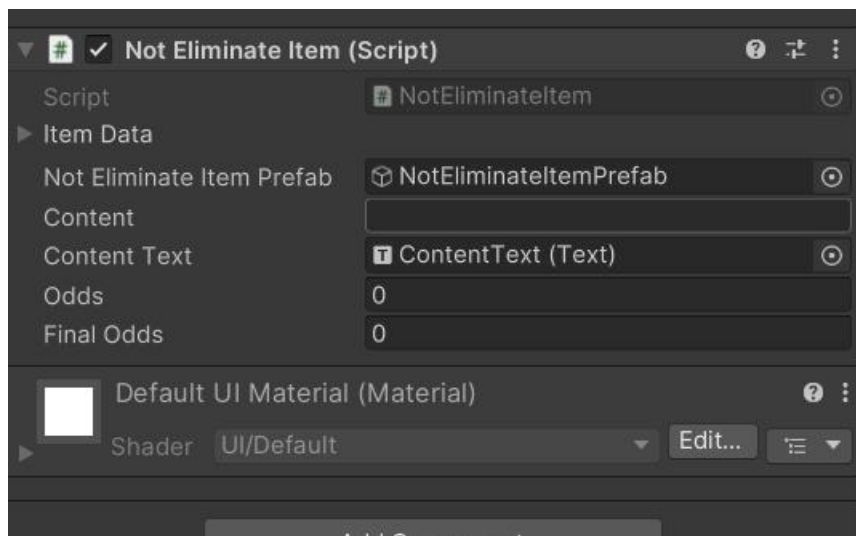


图 4 脚本附着预制件的主要结构

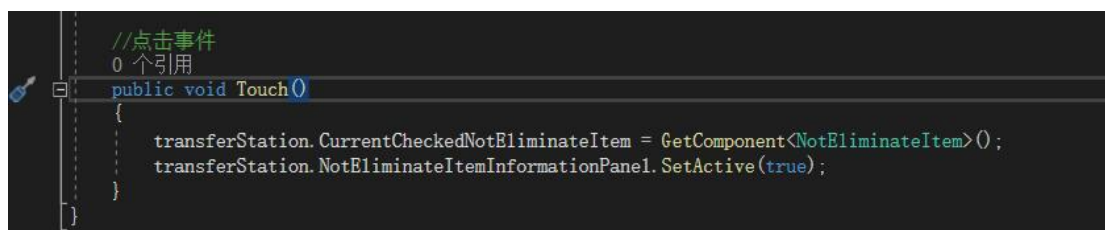


图 5 Touch 点击事件

预制件功能详解：

这个功能是将脚本内的信息，转移到一个全局中转站，然后顺便激活中转站保存引用的 `NotEliminateItemInformationPanel` 面板，这个面板的子级 `InformationPanel` 有个脚本叫 `UINotEliminateItemInformation`，这个脚本附着的物体激活后（父级物体激活，子级物体如果是激活那么也会随之激活；父级物体不激活，那么子级物体一律不激活），将会自动调用 `UINotEliminateItemInformation` 脚本内的 `OnEnable()` 方法，如图 6 所示：

```
void OnEnable()
{
    transferStation = GlobalTransferStation.Instance;
    //找到脚本，然后转gameObject就是物体，根据当前地点找父级
    FatherGameObject = transferStation.CurrentCheckedNotEliminateItem.gameObject.transform.parent.gameObject;
    //Debug.Log("FatherGameObject.name:" + FatherGameObject.name);
    InputField_Content.GetComponent<InputField>().text = transferStation.CurrentCheckedNotEliminateItem.Content;
    InputField_Odds.GetComponent<InputField>().text = transferStation.CurrentCheckedNotEliminateItem.Odds.ToString();
}
```

图 6 从中转站将预制件的信息赋值给信息面板
这样，我们既可以确保修改的选项对象是对的，同时也方便了修改。

信息面板结构物体如图 7 所示：

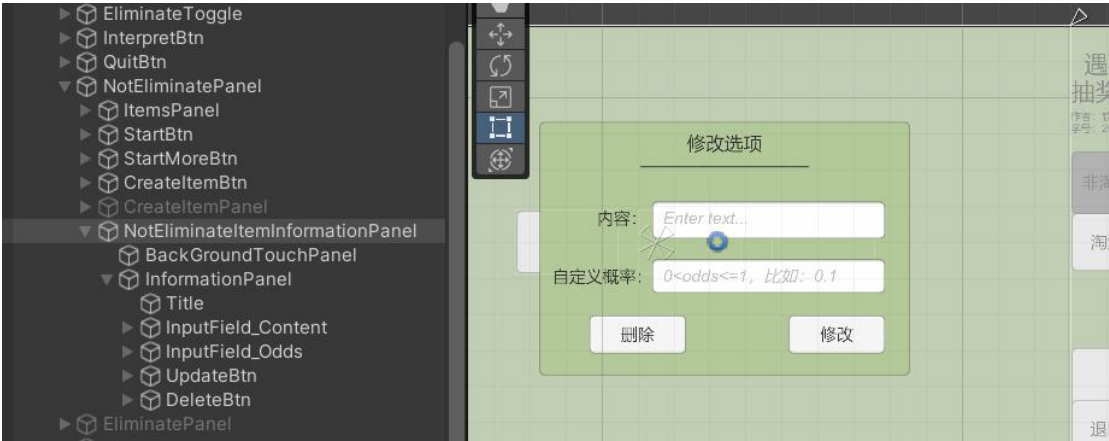


图 7：信息面板结构

信息面板功能详解：

这个信息面板的出现，在创建之后，所以提供的是查看选项信息、修改选项和删除选项三个功能。
查看信息我们上面说了，就是从中转站提取被选中的选项数据，然后直接写入两个 `input` 物体里面。
修改选项与删除选项是对选项的操作，我们后续解释 `NotEliminateManagement` 管理类时会详细介绍，因为涉及到了概率比例重算，以及存档问题。

(2) 淘汰抽奖方面：

淘汰抽奖预制件及脚本基本跟非淘汰预制件相同，除了抽奖的处理外，淘汰和非淘汰其实并

无差别。

唯一区别就是预制件的结构不同，如图 8 所示：

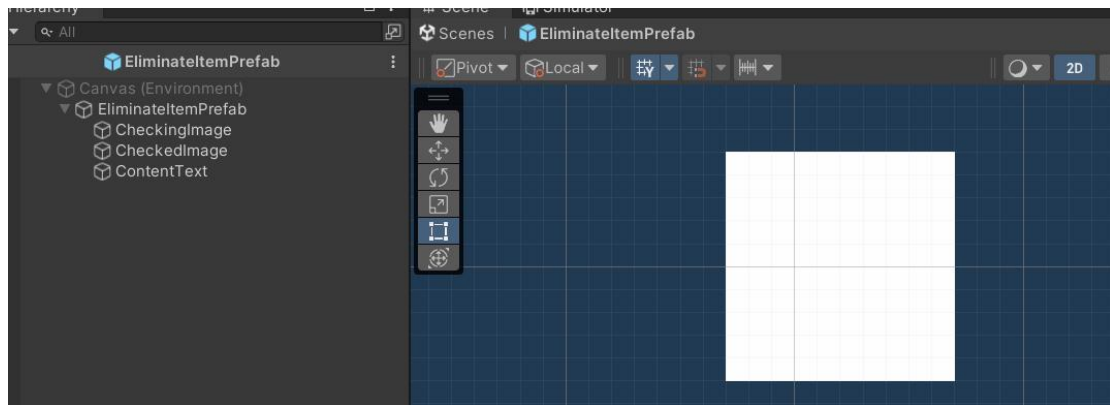


图 8 淘汰预制件的结构

淘汰预制件的结构，比非淘汰预制件多了两个子级物体，两个物体都是图片，这个两个图片都有所含义，并且正常状态下透明度为 0。

当正在选中的状态下，也就是动画轮播到这个选项了，那么这个预制件就会将 **CheckingImage** 物体的 Image 组件的透明度变 1，表示 100%不透明。如图 9 所示：

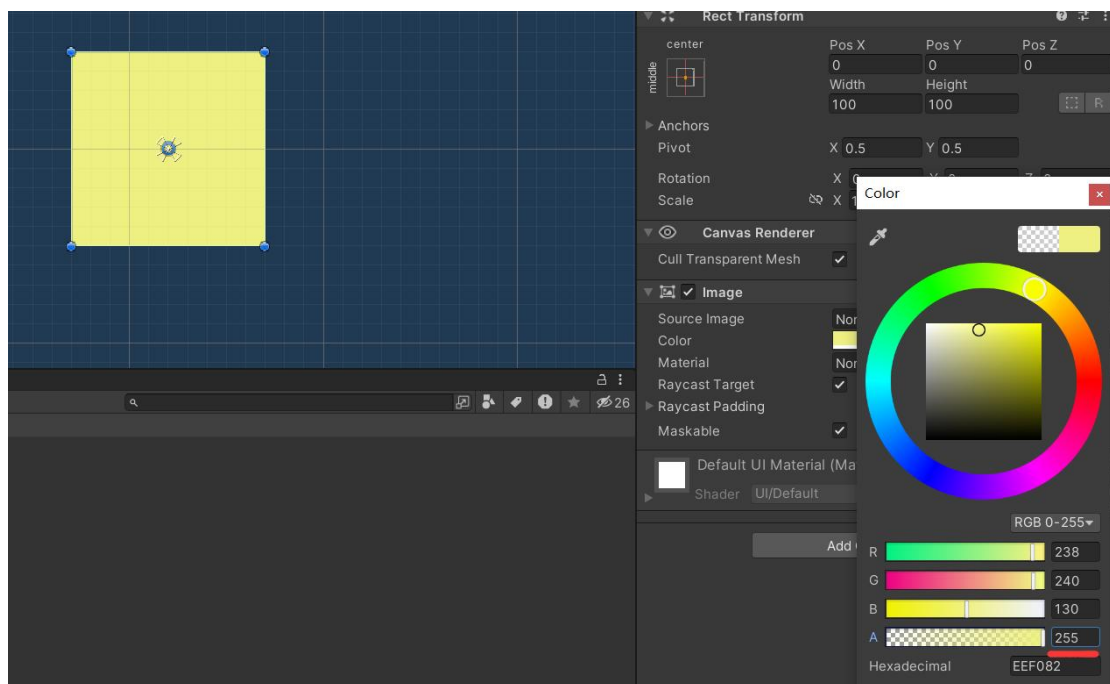


图 9 选项被选的时候

此时透明度 $a = 255$ ，其实就是 1f

如果当这个选项被选了，也就是确认抽到它了，那么就会将 **CheckingImage** 物体的 Image 组件的透明度变 0（严谨一点是这样的），然后再 **CheckedImage** 物体的 Image 组件的透明度变 255，也就是赋值 1f，此时 **CheckedImage** 物体如图 10 所示：

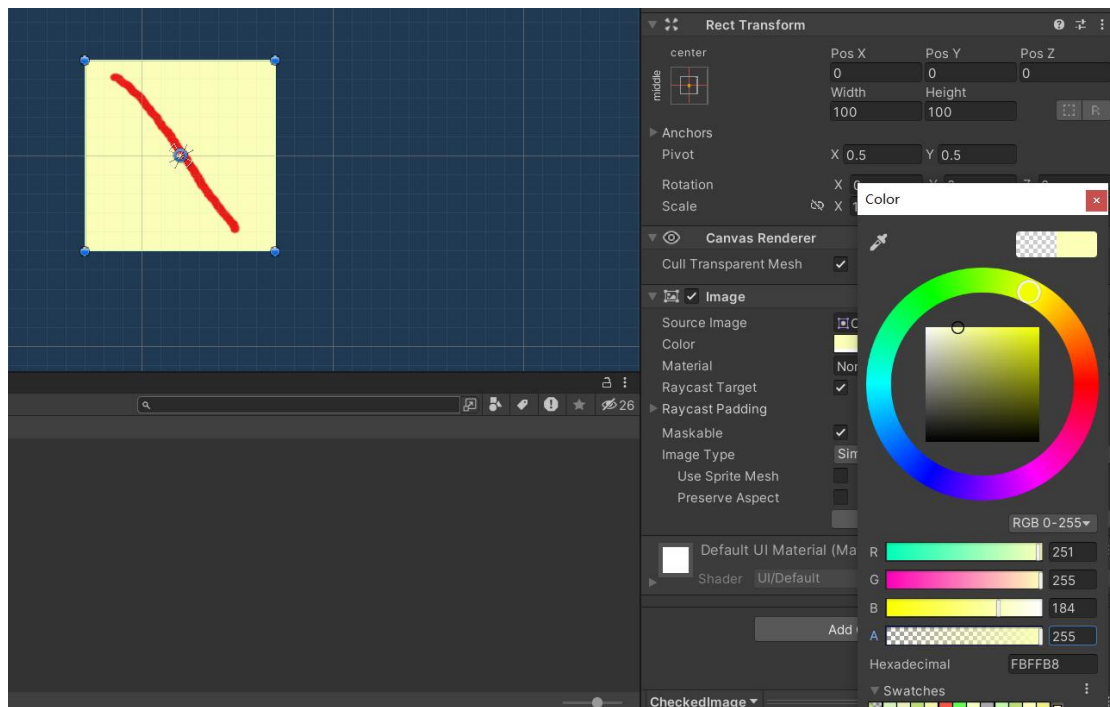


图 10 选项被选了之后

当然被选了，肯定会涉及到一些数据改写，这个我们后面会详细介绍到。

预制件功能详解：

功能跟非淘汰一样提供脚本数据到中转站，供信息面板展示，相应的类，就是非淘汰相关功能的类，其类名去掉 `not` 即可，大体代码功能相同，只是脚本多了俩图片变量，用来控制透明度，也多了一个布尔值 `IsChecked`，然而这个 `checked` 变量是脚本 `EliminateItem` 内对象变量 `itemData`，`itemData` 属于 `EliminateItemData` 类对象变量，这个对象内存在布尔值 `IsChecked`，我们是根据这个对象里面的变量，从而决定是否被选过。

这个额外的类 `EliminateItemData` 是有意义的，它是选项的数据，并且是没有继承任何类，且是可序列化的，我把它当作转化 json 文件的基本数据类，再详细点可以请看第 5 点存档部分。

信息面板功能详解：

淘汰抽奖方面的信息面板，跟非淘汰差不多，只有修改的含义有所区别，修改除了修改内容和概率外，其实也有重置功能，稍微复杂，我们后续解释 `EliminateManagement` 管理类时会详细介绍。

2.首先介绍贯穿全项目的全局单例类 GlobalTransferStation

`GlobalTransferStation` 我一般叫中转站或者全局中转站，或者全局管理类，有可能会混乱文字，所以请谅解，`GlobalTransferStation` 包含多个对象或者变量，是修改，删除，添加的核心类，跟存档功能非常紧密，这个类，我会介绍得非常详细，因为后面提到这个类功能就不解释了。

(1) `NotEliminateItemInformationPanel` 和

`EliminateItemInformationPanel`:

这两个面板变量，是信息面板，点击相应抽奖选项会触发激活这个面板。

(2) `TipMessageText`:

这个提示面板的语句，提示面板的信息，由其他地方赋值到这个变量，如果没有赋值就激活提示面板，那么提示面板回显示错误。

(3) `InititalFontSize`:

这个是初始尺寸，有些时候提示面板字体太大，无法显示全部文字，就会改小文字，这时候就要记住最初的文字，从而利用这个变量。

(4) `NotEliminate_AllOdds` 和 `Eliminate_AllOdds`:

两个不同抽奖类型的分别总概率，这两个变量好像没用过，因为各自抽奖管理类也有相同的变量。

(5) 非淘汰方面:

`ListNotEliminateItem` 为选项脚本的集合，抽奖需要的，`ListNotEliminateItemData` 是选项脚本内置数据的集合，存档功能需要的，`CurrentCheckedNotEliminateItem` 是当前选中的选项，也就是信息面板读取的信息，以及修改，删除，方面要用到。

(6) 淘汰方面的 **ListEliminateItem**、**ListEliminateItemData**、**CurrentCheckedEliminateItem** 与第 5 点非淘汰方面同理。

(7) **RaffleIsRuning**

是淘汰抽奖器动画运转的时候，防止其他操作多次抽奖，bool 值作为开关判定

(8) **createOrLoadData**

它是一个脚本，我们从物体身上获取，它附着的物体是全局最高级别的物体 **Canvas**，这有助于它必然是最先运行的，确保该脚本读取文件赋值，然后让相应抽奖管理脚本去读取信息，创建存档里面的选项。

3. 介绍非淘汰机制的添加修改删除以及抽奖逻辑

现在可以着重介绍一下非淘汰管理类 `NotEliminateManagement`

管理类会事先读取全部固定父级物体整合为一个集合，如图 11 所示：

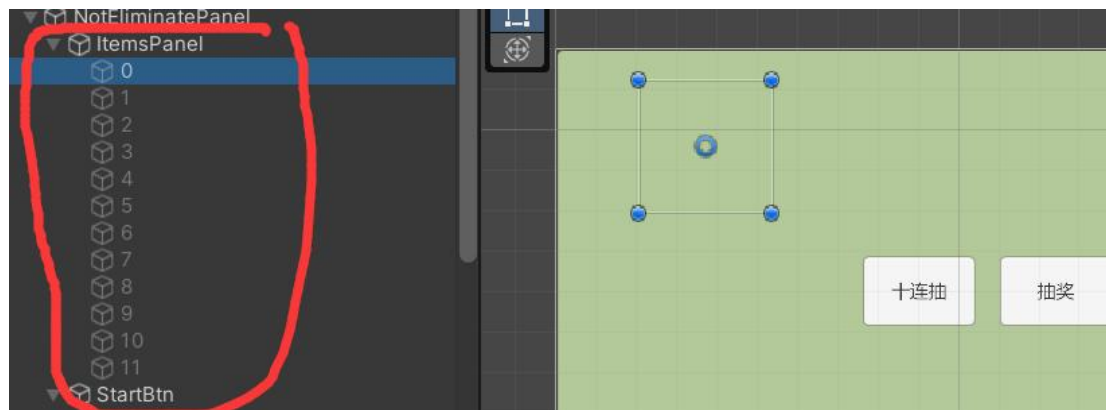


图 11 非淘汰管理类事先读取可以存放预制体的全部父级物体

这个父级物体集合，将会固定每次生成的选项的位置，必须以父级物体为主。

`NotEliminateManagement` 内有最终概率重新分配方法 `SetAllOddsAndFinalOdds()`，供创建选项、删除、修改功能调用，其原理就是：先计算总概率，然后各自选项的概率对总概率的比值，就是最终概率，也可以说是比重。

(1) 添加逻辑（位于 `CreateNotEliminateItem` 脚本）：

- 1) 添加逻辑读取输入的数据，一些规则通过的话，马上实例化一个预制件。
- 2) 随后放在某个未激活的父级物体下面（这需要父级物体集合来每一个判断），同时激活父级物体和预制件。
- 3) 然后调用预制件身上的脚本，调用相应方法赋值，这样才内容才呈现。
- 4) 同时脚本上传到中转站相应的集合。
- 5) 这时候我们要使用 `NotEliminateManagement` 的重新计算方法。
`SetAllOddsAndFinalOdds()`。重新计算最终概率后，我们再调用数据类里面的方法
`SetNotEliminateItemData()`进行数据对象赋值。
- 6) 然后数据对象上传中转站相应集合。
- 7) 然后我们再调用存档方法，写入 json 文件，实现永久保存。完成添加操作。

(2) 修改逻辑（位于 `UINotEliminateItemInformation` 脚本）：

修改逻辑位于信息面板，由于我们打开信息面板，就意味着，我们全局中转站的 `CurrentCheckedNotEliminateItem` 变量已经被赋值了，此时我们可以对这个变量进行一些修改即可。

其实就是添加逻辑的再一次，只是没有实例化预制件这一个过程，因为它本身已经被实例化了，所以我们就是对数据进行覆盖，然后重新计算最终概率，然后调用脚本本身的刷新文本方法，然后调用存档。

由于对象操作，全部都是引用，我们不用再将脚本和数据对象添加到全局中转站的相应集合，因为已经在里面了，且都是同一个对象。

(3) 删除逻辑（位于 **UINotEliminateItemInformation** 脚本）：

删除逻辑一样位于信息面板，由于我们打开信息面板，就意味着，我们全局中转站的 **CurrentCheckedNotEliminateItem** 变量已经被赋值对应对象引用了，此时我们可以对这个变量进行精确删除。删除的逻辑是有规则的，不能变动顺序。

- 1) 先删除中转站集合里面的 **ItemData** 数据（必须要先删除，因为我们使用 **transferStation.CurrentCheckedNotEliminateItem.itemData** 即可调用到，脚本跟数据对象是一体的，唯一性的，如果提前删除 **Item** 集合里面的对应脚本，那么这个 **itemData** 数据，也就是数据对象，就会丢失引用，只能从全局中转站数据对象集合里面一个个搜索，还不一定找的对，因为内容可重复）
- 2) 然后再删除 **Item** 集合里面的对应脚本
- 3) 然后再销毁脚本附着的物体，预制件已经拖拽赋值了这个物体，我们调用即可然后销毁即可
- 4) 然后再将父级物体禁用（空出位置）
- 5) 然后重新分配最终概率
- 6) 然后调用存档保存
- 7) 删除成功后，我们关闭当前页面，完成删除操作

(4) 抽奖逻辑（位于 **NotEliminateManagement** 脚本）：

抽奖逻辑本质就是区间，累加就是区间的另一种方式，就不用单独写最大小值，重复判断。

- 1) 创建随机值
- 2) **sum** 赋 0 值，如果有选项存在，那么进入循环，**sum** 累加第一个选项的最终概率
- 3) 如果随机值小于等于 **sum**，那么随机值在 **sum** 的区间内，那么第一个选项就被选中
- 4) 如果不小于等于，那么 **sum** 就累加下一个，如果不是就继续累加，直到判断成功，那么抽中的就是最后累加的那个选项。

十连抽就是调用十次抽奖，但是抽奖信息单独收集，十次抽完再交付给提示面板

4. 介绍淘汰机制的添加修改删除逻辑以及抽奖逻辑

EliminateManagement 跟 **NotEliminateManagement** 初始功能基本是一致的。区别就在于抽奖的逻辑，以及修改的逻辑多了一些其他功能。

EliminateManagement 的 **SetAllOddsAndFinalOdds()** 方法，比 **NotEliminateManagement** 对应的方法，有了一些改变，因为有些被选了，相应变量已经标识了，那么我们重新收集概率和分配概率的时候，可以直接使用以下代码段跳过这些：

```
if (transferStation.ListEliminateItem[i].itemData.IsChecked)//跳过已经选过的
{
    continue;
}
```

}

这个代码抽奖的时候，也会跳过已选的，确保准确。

(1) 添加逻辑（位于 **CreateEliminateItem** 脚本）：

与非淘汰添加逻辑基本一致

(2) 修改逻辑（位于 **UIEliminateItemInformation** 脚本）：

与非淘汰修改逻辑基本一致。

但是多了一项恢复可选状态，也就是取消被选状态，恢复到可选状态，同时重新计算概率

(3) 删除逻辑（位于 **UIEliminateItemInformation** 脚本）：

与非淘汰删除逻辑基本一致

(4) 抽奖逻辑（位于 **EliminateManagement** 脚本）：

抽奖大体是跟非淘汰功能差不多一致，但是我们是淘汰，淘汰不代表删除，只是不参与，点开信息面板选择修改可以重新加入奖池。除了抽奖的时候，跳过已选的之外，其余跟非淘汰抽奖其实差不多。

当然是有个轮播动画，其实不算是动画，我用的是协程。当轮播到一个选项的时候，我们就让它在这里暂停一段时间，然后再下一个选项，暂停的时间，会随着迭代次数的增加而固定增加。在轮播出现的时候，其实选项已经被选中了，我们做的其实就是让他先慢慢减速，不断增加暂停时间，到达一定阶段后，我们会将暂停时间增加变低，然后开始判断是否到达选项。

关于判断选项，由于内容和概率是可重复性的，不是唯一性的，当然也是可以设置 id 之类的标识，但是我们本身就有个唯一标识，也就是父级物体 0-11 这十二个父级物体，如上 [图 11](#) 所示，非淘汰跟淘汰结构都是差不多的。

这些父级物体以数字编号是有原因的，数字编号的话，是有助于存档位置的准确恢复，这个方面存档部分讲解会更加详细。

由于父级物体的唯一性，我们只需要判断被选的选项，与判断的选项，父级物体的名字是否一致，直接可以判断选中是哪一个。

选中之后会进行以下步骤：

- 1) 多暂停一点时间再显示信息界面，显得不突兀，不然以为是前一个，然后瞬间到下一个。
- 2) 然后脚本内的数据对象调用被选方法，方法里面会将选中的图片显示，同时标记已经被选了
- 3) 然后参与概率重算，由于标记了，所以会被跳过，最终概率重新计算
- 4) 然后保存
- 5) 然后将选项内容提交给全局中转站的提示信息变量

6) 最后打开信息面板，完成抽奖

5. 介绍存档机制（位于 **CreateOrLoadData** 脚本类）

存档机制有点繁杂。

可序列化的两个抽奖的类定义可序列化，`[System.Serializable]`这串代码放在类名前一行，标记可以序列化。

首先介绍 **DataManagement** 类，它是转 json 文件的数据类，而中转站的两个数据类集合，就是放在 **DataManagement** 类里面的，当调用存档的时候，会自动先将中转站里面的数据集合赋值给 **DataManagement** 相应集合，然后再转序列化为 json 文件。

CreateOrLoadData 脚本类，承担创建存档，读取存档，写入存档（保存存档）功能。

以下介绍 **CreateOrLoadData** 脚本类的各个功能：

(1) 介绍创建文件已经创建存档路径

首次打开软件，我们会判断系统文件中目标文件夹是否存在，如果不存在，那么我们就创建文件夹，然后创建一条文件路径信息，保持于一个变量 **DataPath**，此时未创建 json 文件。

(2) 介绍写入存档，即保存存档

写入存档为整个存档机制核心所在，提供一系列规则，保证保存的选项数据完好。

如果是淘汰抽奖调用的写入存档，那么将会对淘汰抽奖方面的选项进行冒泡排序，此时排序的标识就是根据父级物体的名字，字符串转数字，大小判断。

然后再对数据对象集合进行覆盖赋值，这个不用排序，因为选项排序了，直接用选项调用相应脚本重新赋值集合相应下标的数据对象即可，比如图 12 所示：

```
//不用排序也不用清空，直接从transferStation.ListEliminateItem里面直接覆盖赋值。
for(int i = 0; i < transferStation.ListEliminateItem.Count; i++)
{
    transferStation.ListEliminateItemData[i] = transferStation.ListEliminateItem[i].itemData;
    //由于发现，脚本里面的data类，final只赋值了一次，后续不再修改，如果关闭软件再读档，那么概率是不对的，比例将会不对
    //我们重新赋值，因为Item改了，但是ItemData没改
    //趁赋值，我们顺便也修改
    transferStation.ListEliminateItemData[i].FinalOdds = transferStation.ListEliminateItem[i].FinalOdds;
}
```

图 12 淘汰抽奖存档前排序选项后，数据对象集合重新赋值

这样两个集合就完成了顺序，如果不做以上排序，那么动画轮播会出现一些问题：

因为删除操作是精确删除，剩下的选项集合会重新自动排序，如果此时添加，肯定是塞在集合后面，那么如果我们删除了前面的选项，但是选项在集合里面的顺序是靠后的，那么抽奖的时候，轮播会按照集合的顺序进行顺序访问，此时集合是顺序的，但是动画却不是按选项顺序转一圈，而是突然从某个地方跳到另一个地方，因为集合的选项在另一边。

所以排序是必要的。

后面无论非淘汰还是淘汰抽奖，我们都要重新保存最终概率，因为读取存档的时候，我们没必要再重新计算计算概率。

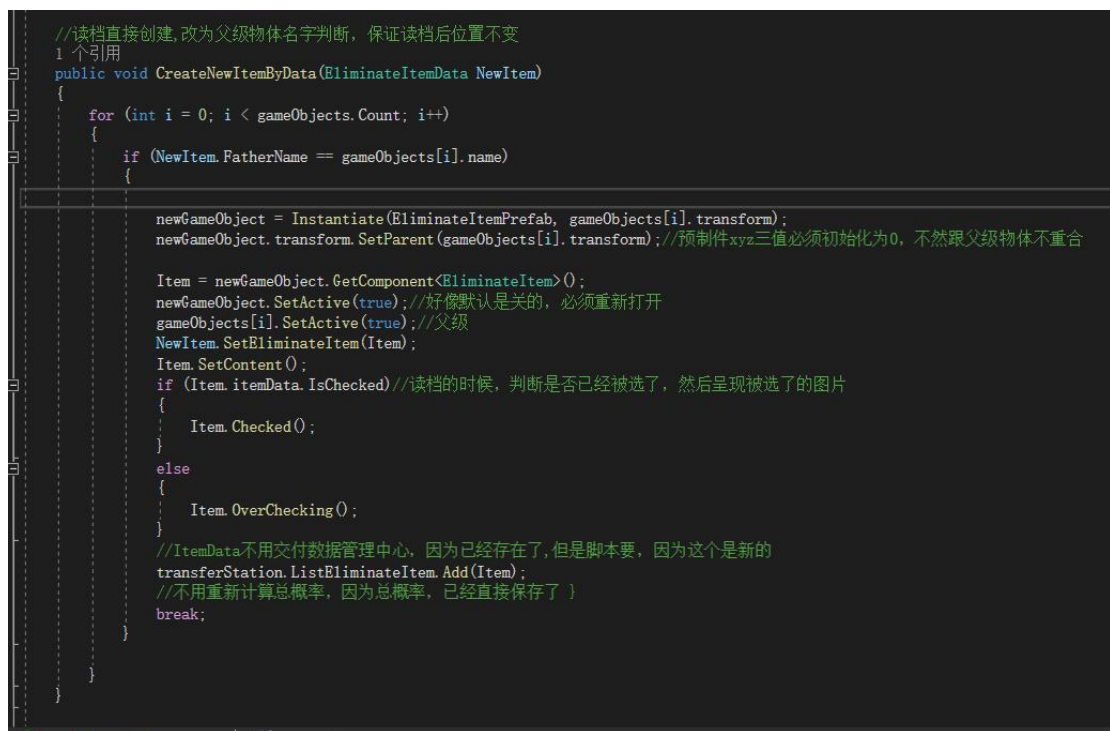
最后创建 `DataManagement` 对象，调用相应方法获取全局中转站对应集合信息，然后根据保存的路径，直接写入 json 文件（如果没有文件，会创建，如果有文件，会覆盖数据）。

(3) 介绍读取存档

读取存档比较简单，如果文件存在，那么就创建 `DataManagement`，从路径变量那里获取 json 文件路径，然后逆序列化为 `DataManagement` 对象，然后调用相应方法赋值给全局中转站对应集合。

(4) 介绍淘汰抽奖管理类怎么在读取存档后顺利恢复选项准确位置和状态

淘汰抽奖管理类读取存档操作，就是在 `Start()` 方法里面调用，如果中转站里面相关集合有数据，那么就循环读取创建，创建类跟添加逻辑差不多，但是如图 13：



```
//存档直接创建,改为父级物体名字判断,保证存档后位置不变
1 个引用
public void CreateNewItemByData(EliminateItemData NewItem)
{
    for (int i = 0; i < gameObjects.Count; i++)
    {
        if (NewItem.FatherName == gameObjects[i].name)
        {
            newGameObject = Instantiate(EliminateItemPrefab, gameObjects[i].transform);
            newGameObject.transform.SetParent(gameObjects[i].transform); //预制件xyz三值必须初始化为0,不然跟父级物体不重合

            Item = newGameObject.GetComponent<EliminateItem>();
            newGameObject.SetActive(true); //好像默认是关的,必须重新打开
            gameObjects[i].SetActive(true); //父级
            NewItem.SetEliminateItem(Item);
            Item.SetContent();
            if (Item.itemData.IsChecked) //存档的时候,判断是否已经被选了,然后呈现被选了的图片
            {
                Item.Checked();
            }
            else
            {
                Item.OverChecking();
            }
            //ItemData不用交付数据管理中心,因为已经存在了,但是脚本要,因为这是新的
            transferStation.ListEliminateItem.Add(Item);
            //不用重新计算总概率,因为总概率,已经直接保存了
            break;
        }
    }
}
```

图 13 从存档里面读取创建选项，循环调用该方法

跟添加逻辑差不多，但是我们选项脚本里面的数据对象记录了父级物体的名字，通过名字，我们可以判断出，应该实例化预制件在哪个父级物体下面，这样确保了选项的位置准确，然后我们再判断数据对象里面的标识，是否已选了，然后分别调用相应方法。非淘汰抽奖管理类也是基本相同的操作，我们就不进行介绍了。

四、未来可继续优化方面

1. 非淘汰抽奖的算法优化

非淘汰抽奖其实我预优化了，由于是累加判定，我是打算使用前缀和+二分查找的，如图 14 代码所示：

```
//重新计算总概率，通过添加选项和删除选项调用。  
3 个引用  
public void SetAllOddsAndFinalOdds()  
{  
    if (transferStation.ListNotEliminateItem.Count != 0)  
    {  
        //计算总概率，必须先将AllOdds归0  
        NotEliminate_AllOdds = 0;  
        for (i = 0; i < transferStation.ListNotEliminateItem.Count; i++)  
        {  
            NotEliminate_AllOdds += transferStation.ListNotEliminateItem[i].Odds;  
        }  
        Debug.Log("NotEliminate_AllOdds:" + NotEliminate_AllOdds);  
        //通过比例的情况，重新分配最终概率  
        for (i = 0; i < transferStation.ListNotEliminateItem.Count; i++)  
        {  
            transferStation.ListNotEliminateItem[i].FinalOdds = transferStation.ListNotEliminateItem[i].Odds / NotEliminate_AllOdds;  
            Debug.Log("分配完FinalOdds:" + transferStation.ListNotEliminateItem[i].FinalOdds);  
        }  
  
        //同时计算前缀和，方便十连抽，直接得出结果，而不是重新叠加计算  
        /* PrefixSum = new double[transferStation.ListNotEliminateItem.Count];  
        PrefixSum[0] = transferStation.ListNotEliminateItem[0].FinalOdds; //先确定第一个数  
        for (i = 1; i < transferStation.ListNotEliminateItem.Count; i++)  
        {  
            PrefixSum[i] = transferStation.ListNotEliminateItem[i].FinalOdds + PrefixSum[i-1];  
        }  
        //测试  
        for (i = 0; i < PrefixSum.Length; i++)  
        {  
            Debug.Log("PrefixSum[" + i + "]: " + PrefixSum[i]);  
        }  
    }  
}
```

图 14 前缀和预处理

但是最终被我注释掉了，因为选项最多也才 12 个，累加法我试了连续抽一千次，也是瞬间得出结果，如果后面有需求，可以再重启启用这段代码。

当然淘汰抽奖就不用使用前缀和了，因为选项并没有消失，而是隐藏，不好处理二分查找。

五、结尾总结

遇事不决抽奖器，做了四天，大概两千行左右代码，收获确实满满。

之前做过第一版的，可惜项目误删了，只记得思路。这个安卓作业让我顺便把他翻新再做一遍，以前版本可没有自动存档功能，界面也没有那么好交互，借此机会重现，发现对比，自己开发水平跟以前已经不是一个量级了。

当然缺点也是很多的，缺乏趣味性，确实略感无聊，单独是这样的，如果能跟其他游戏项目合并，淘汰抽奖添加条件，每次抽完，条件变多，很上头的。