游戏逻辑思想

Game Logic Thinking

作者：陈健

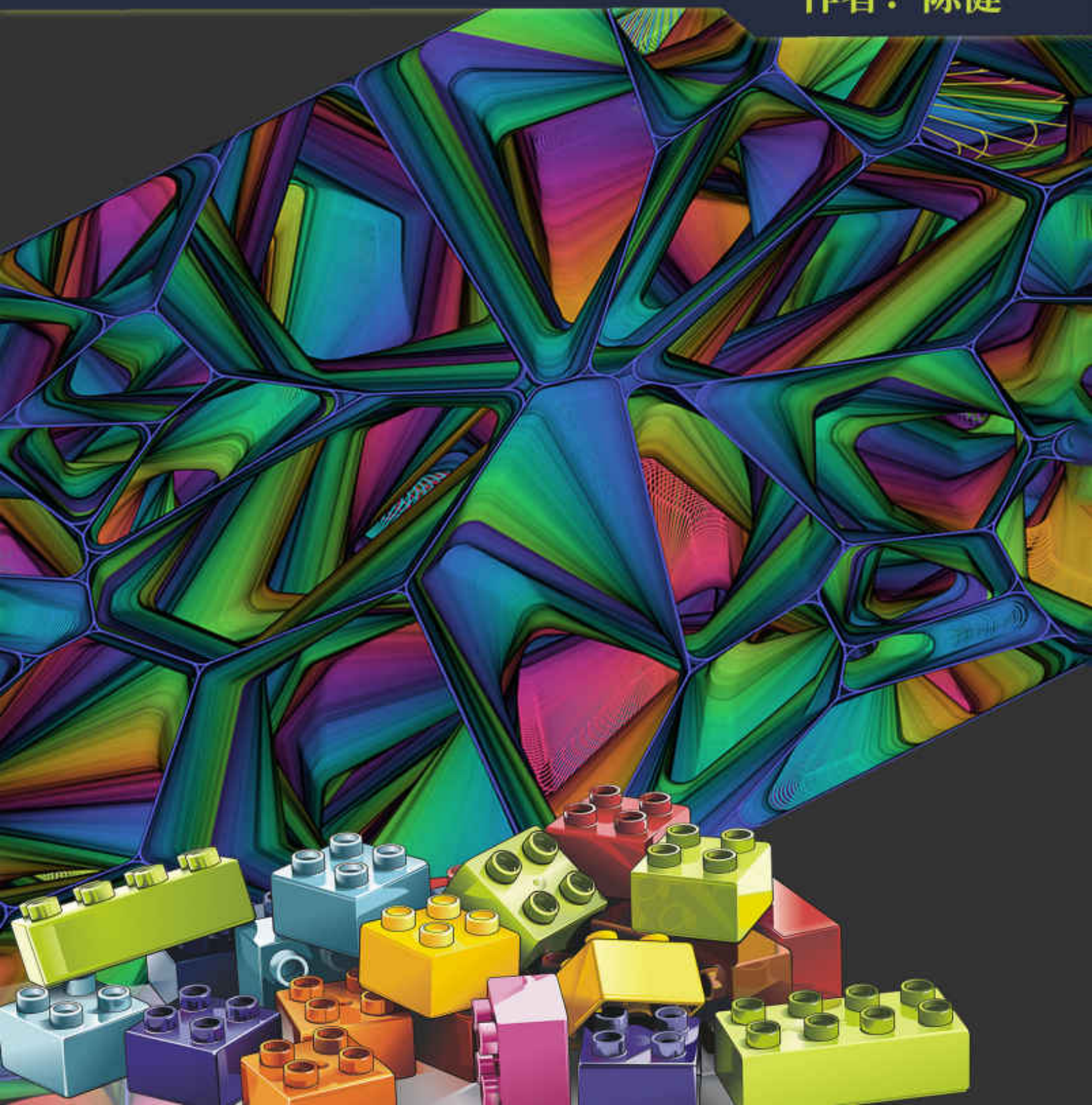


Table of Contents

游戏逻辑思想

版权信息

游戏逻辑思想

DNA-BN:ECFD-NO0017157-20190711

出版：浙江出版集团数字传媒有限公司
浙江杭州体育场路347号
互联网出版许可证：新出网证（浙）字10号
电子邮箱：cb@bookdna.cn
网址：www.bookdna.cn
BookDNA是浙江出版联合集团旗下电子书出版机构，为作者提供电子书出版服务。
如您发现本书内容错讹，敬请指正，以便新版修订。

目录

序

这本书是讲述游戏开发过程中逻辑部分的合理实现以及实现方式。内容涵盖前端与后端的逻辑设计模式，以及框架的编写。本书尽可能少讨论细节的实现，更多的是讨论设计，框架，以及如何更好的实现。

本书希望能：

1. 引发更多的启迪

2. 减少与策划之间的沟通成本

3. 减少重复的代码量，减少代码产生的错误

4. 享受编程本身的乐趣

策划经常会说：这个功能或者界面与那个一样，直接复制一份。他们也经常配置错误，导致进程无法运行。

程序经常说：天天写逻辑，非常没意思。也会经常说：我这个实现也没问题，为什么一定要按你说的方式写。

美术经常说：我这个文件要放哪里，放这里可以吗？也会经常说：这个效果与效果图不一致，你看那个系统的表现就是对的。

测试经常说：这个操作就会崩溃，那个界面就不会。也会经常说：系统开发完有这么多bug是正常的，修几天就好了。

运维经常说：崩溃了，程序看一下。

运营经常说：玩家的充值不到账，注册数据不对了。

老板说：这周的内容A,B,C,D…

很多很多的项目问题，很多的bug，很多的灰色地带需要产生大量的沟通成本。

大量的沟通后，然后开始建框架，评估设计，定方案， 交流，用技术手段规避上面的问题，甚至是节约大量的沟通成本，这里面大部分的东西都是基于逻辑思想来完成的。

第一章

基础内容交流

1.代码规范，编写可读懂的代码

代码有许多种命名的方法，比如驼峰命名法、匈牙利命名法、帕斯卡命名法和下划线命名法。这些方法是语言级别的命名方法。

驼峰命名法除第一个单词之外，其他单词首字母大写。比如下面的代码：

```
int myStudentCount;
```

我们在其他地方看到myStudentCount的时候，我们可以很快的通过它的英文或者拼音理解它的含义。但是重要的是，我们无法一眼知道它的类型。

下面我们讨论动态语言的代码规范，因为动态语言的类型灵活性，它的代码比其他语言要严格的多。我们以lua为例，讨论命名的规范性。

首先，是变量的命名。

字符串统一以s开头，比如 sText，s意味着string

整形或者浮点统一以n开头，比如nCount，n意味着number

布尔类型统一以b开头，比如bShowMsg，b意味着boolean

数组或者map统一用t开头，比如tAward，t意味着table类型或者map类型

函数对象统一用f开头，比如fCallBack，f意味着function

对象统一用o开头，比如oButton，常用在更为底层的对象上，o意味着object

从上面可以看到，我们的命名应该带上足够的信息，这些信息应该包含了它是什么类型。在类型方面，又存在2种不同的类型条目，程序类型以及逻辑类型。我们前面已经定义了程序类型，接下来我们可以对逻辑的命名进行拓展。比如对于某个库的ui对象的命名。

假设这个ui的类型在该库中是Label。那么我们可以把它命名开头定位oLbXXXXX。最后，我们给按照功能给这个对象进行命名。比如这是一个标题栏的Label,那么我们可以命名为oLbTitle。这里就会有一个疑问，为什么要把类型放在前面而不是把作用放在前面，比如说命名为oTitleLb。这是因为逻辑功能的命名通常长于类型命名，类型放在后面不容易快速识别是什么类型，而放在前面就很容易可以识别出来了。所以我更建议把类型放在前面。

以get开头+名词的函数，在函数内不能对对象或者现有环境产生副作用。经常遇到的一种情况：就是为了使下次的get能够查找的更快，对对象进行缓存。这种接口也不能直接使用get 开头+名词的方式。

如果在get中希望也做一些事情，那么命名的接口为getAndDoSometing。get接口的返回值应该非常清晰，直接就对应着命名，比如上面的问题可以命名为getAndCacheXXX。在get开头+名词的接口的参数中传递一个回调也是不应该的，比如getXXX（fCallaBack）。fCallBack里面本身可能会对整个环境做修改，所以也需要用getAndDoSometing。

以set开头的函数，一定是对变量进行直接的赋值，不能是加或者减。加或者减的操作应该使用更加清晰的add，sub接口。set接口的返回值可以是无类型或者是boolean，返回boolean常用在判断是否真正产生了变化。这种情况一般是这么写的：

```
function setFlag(bNewFlag)

    if self.bFlag == bNewFlag then

        return false

    end

    self.bFlag = bNewFlag

    --做一些别的操作

    return true

end
```

如果支持多返回值，那么返回值可以带上一些修改后的值。

注意在set接口中一般需要判断是否真的改变，只有真的发生改变了才执行后面的逻辑。如果在部分场景需要强制赋值，那么有2种方式：

1. 添加一个tOption参数，通过该参数对不同的可选参数进行不同的处理，这边的可选参数是个map类型或者说是个table，所以用t开头。

2. 新增一个直接赋值的接口，该接口内不进行比较判断。原来的set接口改为判断后调用新的接口。

我们来欣赏一段赋值代码，取自白鹭引擎的底层代码Bitmap.ts，来加速我们队赋值函数的理解。

```
$setTexture(value: Texture): boolean {

    let self = this;

    let oldTexture = self.$texture;
```

```
        if (value == oldTexture) {
            return false;
        }
        self.$texture = value;
        if (value) {
            self.$refreshImageData();
        }
        //省略其他代码
    }
```

我们在代码中，经常也会看到以on开头的函数，这类的函数主要是用来作为事件响应的。比如关闭按钮的响应事件onClickClose，又或者是响应其他对象发出的事件，如onLevelChange，注意这类的接口外界看到是不应该直接去调用的，即使它不小心把自己设置成为了public。

在我们的一些语言中，比如lua之类的动态类型的语言。我们无法通过ide来限制外界对内部的访问。那么我们需要建立一个项目共识，比如带下划线的函数是禁止从外界调用的。带下划线或者$开头的成员变量也是不能直接去访问的。总结起来就是所有看似是私有的函数或者变量，都不应该直接去访问。当然，变量应该只通过接口去访问。

一个常见的场景，比如项目中已经有一些代码与你遵循的代码规范不一致的时候，这时候我们建议的是在原作者的文件中，遵从原作者的写法。在自己新建的文件中，按照自己的规范来写代码。还有一种场景，如果你使用的框架或者第三方库与你遵循的代码规范不一致，这时候你依然按照自己的规范书写代码，除非你是直接去修复框架或者第三方库的问题。

接下来讨论另外一个问题，成员变量的放置问题。我们通常在别人的代码中看到的是把成员变量定义在一个类最开始的地方。我们这边提出另外一种可能性，就是把成员变量放在对应的函数前面。比如下面这种：

```
    private bFlag = false;
    public getFlag(){
    }
```

```
public setFlag(bFlag){

}
```

什么时候我们把变量放在函数前面呢？当这个变量的有效赋值以及使用只有部分函数，且这些函数都是紧紧挨在一起的时候，我们把它放在它的前面。这是遵从一个更大的原则，有关联性的东西应该放一起。

2. 正确的使用断言与返回

有一天的清晨，老A早早的来到了办公室，决定对团队的成员近期的代码进行一次code review（代码审查）。当然，审查对象是入职不久的小C。小C打开svn，展示了一段最近写的代码。

```
function processItem(){

    assert(gItem) //断言道具表存在

    assert(gEquip) //断言装备表存在

    //其他代码

}

function processItemEquip(){

    if (! gItem) {

        return

    }

    if (! gEquip) {

        return

    }

    //其他代码

}
```

这2个函数的意图很明显都是在函数入口处对配置不存在的时候进行的处理。那么我们首先应该清楚在逻辑上，这2个配置表可能是为null的，也就是空的。当配置表的值为空的时候，我们去访问它的内容就会报错，所以我们需要去处理这个情况。那么我们究竟应该选择哪一种写法呢？

先讨论一个问题，什么时候应该用assert进行断言，什么时候应该用return。

a. 在逻辑上面允许发生的情况下，我们应该用return。

b. 如果是逻辑上认为不应该发生的，我们应该用断言。

在上面的问题上，如果在调用processItemEquip可能存在gItem为空的情况，那么我们在这边应该用return。如果在设计上，processItem调用的时候gItem应该存在了，那么应该用assert。我们不希望我们在使用一个配置表的时候都需要去判断它是否为空。所以当我们应该把逻辑设计成在一个函数调用某个配置的时候，这个配置就是存在的，这样会大大减轻后续代码的各种判断。如果配置在访问的时候就应该是存在的，那么它一定不会是空的。如果它是空的，那么一定是我们的设计出现问题。这时候我们要大胆使用assert，在第一次出现问题的时候及早中断执行流程，立刻排查错误。

讨论下面一个场景，写一个引用计数的2个接口

```
this.nRef = 0;

function incRef(){

    this.nRef = this.nRef + 1

}
```

下面是2种写引用计数的写法，哪种更正确？

```
function decRef(){

    this.nRef = this.nRef  -  1

    assert(this.nRef >= 0)

}

function decRef(){

    if(this.nRef <= 0){

        return;

    }

    this.nRef = this.nRef  -  1

}
```

第一种是断言，第二种是容错。

小C觉得使用第二种比较好，这样就没有必要关注外界用错导致引用计数被多减的情况，方便了外界逻辑层的调用。那么为什么我们还是推崇第一种写法，因为第一种写法是

在问题发生的时候第一时间把错误断住了，它要求逻辑调用者必须找到引发断言的根本性问题。

那么我们经常在什么时候会写return这种兼容的代码？比如说一个网络消息过来，检测道具数量够不够，第一次协议过来是够的，第二次协议过来就不够了。那么我们在检测道具不足的时候用return而不是assert。那你可能会说客户端也加检测就可以assert了。考虑到第一条协议可能还没返回新的数据给客户端的时候第二条协议也过来(数据层我们统一以服务器为准），这时候客户端是拦不住的，所以这种情况我们使用return。

3. 可拓展接口设计

在项目的开发过程中，接口可能会经历不同的需求，可能会越来越糟。比如下面的情况：

```
addModulePower(nModuleId, nPower);
```

经过一系列的需求变更，可能变成下面这样：

```
addModulePower(nModuleId,nPower,bSyncMsg,bSendEvent, bOnlyBoss)
```

这个接口经过各种需求的历练，加了三个参数。我们先不讨论这3个参数是否合理，当做是需求必须要加的。

对于addModulePower这样的接口，我们可以很明确的得出，它至少需要2个参数，nModuleId和nPower。那一个大的项目，它可能需要增加各种各样的参数，根据不同的参数在函数内部使用不同的计算方式。这时候我们应该是用Options这个东西了，Options在lua里面就是一个表，在typescript里面就是字典，我们将接口改成：

```
addModulePower(nModuleId, nPower, tOption)
```

以后所有的拓展接口都需要在tOption里面去加，保证了接口的干净。在函数的内部进行

```
if( tOption && tOption.bSyncMsg){

    //做这个参数该干的事情

}
```

这边我们总结一下接口参数应该满足的特点：

1. 接口应该包含跟命名一致的参数，并且按照参数的重要程度依次进行定义

2. 接口应该具备可拓展性，在不同的需求下应该都是来去自如的进行功能的拓展。

这边提到的tOption这个东西，在很多地方都可以看到它的影子，比如说在微信小游戏的云开发里面：

初始化

在小程序端开始使用云能力前，需先调用 `wx.cloud.init` 方法完成云能力初始化（注意小程序需先开通云服务，开通的方
此，如果要使用云能力，通常我们在小程序初始化时即调用这个方法。

`wx.cloud.init` 方法的定义如下：

```
function init(options): void
```

`wx.cloud.init` 方法接受一个可选的 `options` 参数，方法没有返回值。

`options` 参数定义了云开发的默认配置，该配置会作为之后调用其他所有云 API 的默认配置，`options` 提供的可选配置如

字段	数据类型	必填	默认值	说明
env	string | object	否	default	默认环境配置，传入字符串形式的环境 ID 可以指定所有服的默认环境，见下方详细定义
traceUser	boolean	否	false	是否在将用户访问记录到用户管理中，在控制台中可见

图1

还有它对应的数据库的初始化：

wx.cloud.database

获取数据库的引用

方法签名如下：

```
function database(options?: object): Database
```

方法接受一个可选对象参数 options，其字段定义如下：

字段名	类型	必填
env	string	否

图2

从图1图2可以看到，这里init接口的参数只有一个可选的options，并没有其他参数。带有options的接口是对拓展性非常友好的。

那么我们会产生一个问题，是不是所有的接口都应该改为只接受一个options参数。从大量的代码可以得出结论，并不是这样的。我们还是要将参数依靠经验以及逻辑划分出较为固定的和可变的部分，只有可变的部分才使用options进行拓展。在addModulePower(nModuleId, nPower, tOption) 这个接口的设计我们可以看出，nModuleId, nPower就是相对固定的部分。

这边涉及到一个方法论的问题。在美术培养审美的过程中，他们会被要求多看一些厉害的人画的作品，从而提升美感。对于程序而言，也是一样的，也要多看一些别人的设计。通过对这些设计进行抽象以及总结，来提高自己对于代码的驾驭能力。我们也可以通过对比，来找到合适的接口形式。比如下面这个接口：

CreateWindow("Main","Windows",WS_OVERLAPPEDWINDOW,200,200,500,500,NULL,NULL,hInstance,NULL);

我们发现经常在使用这个接口的时候传递了很多的NULL，这就是一种不舒服的调用形式，尤其容易错位。我们应该尽可能的使用可拓展的接口形式，使用者只需要关注自己需要的特性即可。

4. 调试的思维与逻辑

当代码文件数量变动，代码量也跟着变多的时候，很容易出bug，有bug的地方就需要调试。

首先要在bug发生的时候我们要先观察它的表现，如果有日志或者出错信息的直接看出错堆栈。如果没有的情况下收集案发现场表现，进行初步排查。这边很重要的是要练习去猜测问题的答案，这方面主要是结合最近团队成员的代码变更以及对项目本身代码的熟悉程度进行。

对于可以重现bug，通过多次重现收集它的信息，将问题进行2分缩小，直到问题被最终定位。对于不可重现的bug，思考如何在发生异常的时候及早将错误的代码断下，可以使用我们前面说过的asset进行断言。

分析过程主要依赖于2个思维：

正向思维

逆向思维

举个简单的例子，点击一个按钮产生了一个不在预期的飘字。

对于正向思维而言，我们可以在代码中找到按钮对应的响应事件，从响应事件中找是否有飘字的代码出现，递归直到找到该飘字。阅读代码主要通过命名的识别以及关键的数据结构进行快速浏览，配合函数引用以及文本查找。

对于逆向思维而言，我们可以在飘字的地方下断点，当重复操作使该飘字代码断下后，沿着堆栈找它是在哪里被调用的。

在一个bug的处理过程，我们需要交叉使用正向和逆向思维去排查问题。通过正向思维去简单获得bug的上下文信息，通过逆向思维去排查。简单或者少量的代码（300行左右的代码问题），建议直接采用正向思维。对于复杂的代码，建议2种思维并用。

那是个发版前的夜晚，小A一直愁眉苦脸的坐在电脑前，看着一个一个代码文件和禅道的一个bug单，他还是妥协了，决定找老A帮忙解决问题。有个问题是这样的：在一个窗口使用了一个道具，正常情况使用后对应的显示道具数量的地方就会发生变化。但是使用后，窗口上面的道具数量还是没变。

老A坐到小A的位置上，准备复现看看什么问题。老A使用命令添加了一个道具，使用后，正如小A的描述，对应系统界面的道具数量没有变化，老A再点了一次使用，客户端提示：该道具数量不。很明显，老A在收集案发现场中的有用信息。老A立刻断定数据层是对的，因为第二次点击的提示是对的上数据本身的变化的，所以这个问题应该是界面问题。

将问题锁定在界面之后，老A找到数据层变更时界面注册的响应。在响应函数中下个断点，再次重复操作重现这个异常表现。断点被断住了，说明这个事件已经被窗口响应到了。那么问题就进一步缩小在这个响应代码里面了。这时候该正向思维出场了，快速阅读代码，发现这个道具可能触发不同的刷新行为。当出现分支的时候，在代码中也就是if else这样的东西，通常我们要高度警惕，bug往往就因为执行流程走了其他的分支导致。这时候我们可以使用正向思维也可以使用逆向思维继续排查。对于正向思维，我们要在断点之后一步一步执行代码流程，专业术语叫单步执行。看看在哪个分支代码执行流程与我们的设想不符。对于逆向思维，我们可以在所有的if else分别下断点，看看断下的地方哪个不是我们预期的。当分支很多的时候，如果没有新的信息支撑我们下断点，我们就需要下大量的断点且可能会遗漏一些分支的执行流程。这个时候我们比较推荐的是使用正向思维，它有助于更多的收集信息。

从上面的例子我们可以看到，逆向思维的排查速度更快，依赖于有效的断点信息。正向思维排查速度慢，但有利于收集到更多的代码信息，加深对代码的理解。所以我们总是切换着使用它们。

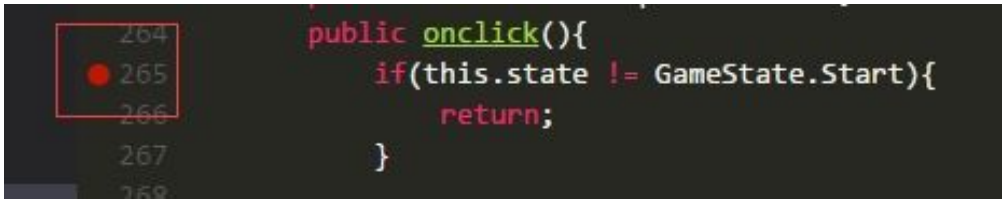
我们再来看另外一个案例。小C跟着团队转去做h5，开始学着使用白鹭引擎。在ui的上手过程中，他遇到了一个问题。有A，B两个界面，B在A的上面。小C希望B界面不要响应而A界面正常响应。大部分的时候是正常的，但是当B界面上面有个图片的时候，A界面就无法得到响应了。小C严重怀疑那个图片把响应事件吞了，他把该图片的touch也及时触摸相关的设置都设置为了false，也就是不响应触碰事件，但依然有这个问题。这时候小C就懵逼了，于是只好求助老A。

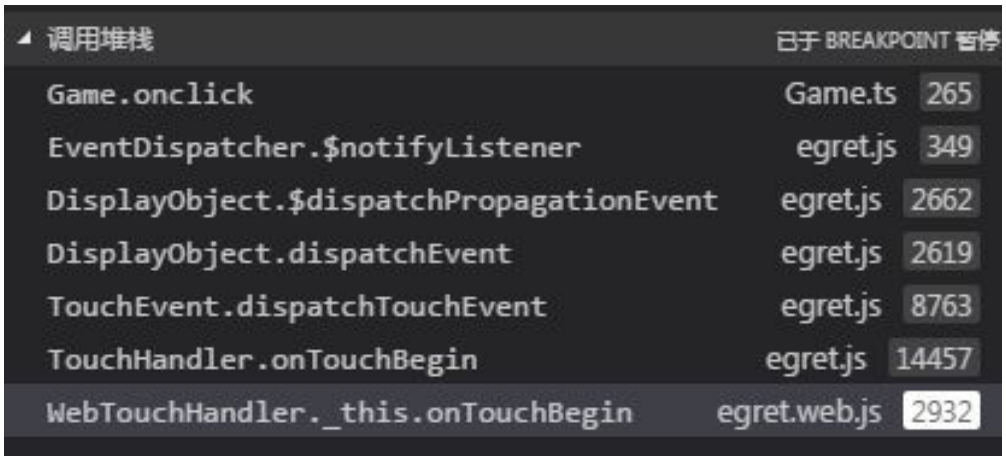
老A听了小C的描述，没有立刻去找问题，他先确定了小C所描述的情况都是正确的（如果描述者的性格是严谨的，那么我们可以花较少的时间去确认当前情况。如果不是，我们第一步就要先验证现象）。通过确认，属性以及相关设置都是对的。小C怀疑可能是引擎问题，立刻去升级了测试。鉴于这是个基础问题，所以老A觉得引擎不会也不应该犯这种基础错误，所以他怀疑是别的地方，这是个经验判断。

重新审视了现有的条件，存在正常的点击和不正常的点击。那么，我们可以通过正常点击，并使用逆向思维在函数响应的地方下断点获取到正确的调用路径（堆栈）。在调用路径上依次下断点，重复不正常的点击。哪个断点没有触发，就是对应的分支出了问题。

整个操作过程如下：

先在正常的A界面的响应里面下个断点（如下图所示）：





断点断住后获取到堆栈（如下图所示）：

堆栈上有7个函数，从上往下，依次在调用的地方下断点。然后再点击B界面上面的图片，看看哪个断点没有正常触发，没有正常触发的地方就是异常的地方。定位到异常的地方后，再次使用正向思维，可以快速的找到异常的根本原因。最终，这个问题的原因是触摸事件被别的对象截获了点击事件。这是逆向思维为主导的一个应用。

在我们的日常开发中，要提高调试能力实际上是不容易的。大量的错误，大部分的重复性问题，比如说打字错了，忘了窗口是异步加载等。调试能力的训练需要在我们完成手头开发任务的情况下，主动去禅道或者项目bug的平台上去找一些自己认为能提高能力的单子。主动反复通过这些单子去训练，可以慢慢的积累解决bug的推理能力。当然，最终的提高是通过总结这些bug的来源，在系统上提出解决方案来规避它的出现。解决只是手段，规避是核心。主动去接单修复是提高bug调试能力的一个重要态度。

极端的bug，也就是很多程序程序口中诡异的bug。这类bug复现比较难，我们可以通过输出日志的方式来记录它的行为。在下次发生的时候，通过这些输出来进行推理，就像柯南破案一般。但是这样依然可能没有办法获得有效信息。这时候要做的就是细化，细化我们的输出，达到输出信息可以有效进行分析为止。所谓的可以有效分析，需要输出这个bug发生时候的上下文，包括局部变量，对象成员，甚至是全局变量。通过这些信息来尝试使用逆向思维。如果依然无法解决，那么就转用正向思维来收集信息。

5. 培养敏锐的异常反应

我们经常在老程序口中听到对称、平衡、泄漏等词汇，如果不注意就容易出现内存泄漏或者其他问题。对称，指的是类似于创建对应销毁，申请对应释放。在C/C++开发的程序中经常会被提到要做好内存的申请与释放。在unity引擎开发的程序中，会被提到要注意资源的使用与释放。关注对称是值得我们培养的意识。那么，不对称意味着什么？

我们先来看一段底层框架日志的输出

```
----------------------

OnCreateWindow Test1

OnCreateWindow Test2

OnDestroyWindow Test2
```

这个日志输出简单的解释可以理解为先创建了Test1窗口，紧接着创建了Test2窗口，随后销毁了Test2窗口。

那么假设对应的界面为下面的情况：

1. 只在界面上看到过Test1

这意味着Test2 是在同一帧进行创建和销毁

2. 界面上出现Test1，接着出现Test2，随后关闭了Test2，关闭了Test1这意味着底层框架出现了问题，底层框架并没有办法在任何情况都做到在窗口关闭的时候捕获销毁事件

我们再看下另外一种日志：

```
----------------------

OnCreateWindow Test1

OnCreateWindow Test1

OnDestroyWindow Test1
```

这个日志输出简单的解释可以理解为 先创建了Test1窗口，紧接着再次尝试创建了Test1窗口，随后销毁了Test1窗口。

那么假设对应的界面为下面的情况：

1. 在界面上看到过Test1，点击了一次后Test1被销毁了

这意味着底层防止了创建相同的窗口

2. Test1创建，并没有被销毁

这意味着底层可能使用引用计数管理界面，只有界面引用计数为0的时候才销毁窗口

3. 在界面上看到过Test1，点击了一次关闭后Test1还在

这意味着底层允许创建同类型的窗口，并且区别对待每个实例

我们尝试把一个生命周期里面（对应上面就是在界面上看到窗口创建和销毁的过程），出现的AAB型日志或者其他AB数量不平衡的日志称为不对称日志。这种日志都代表了异常或者发生了bug。当然，我们这么说的前提是我们的设计不能以容错来设计，参考断言与返回这一节。那么在这种非容错的设计下面：

```
------------------------------

OnCreateWindow Test1

OnCreateWindow Test2

OnDestroyWindow Test2

OnDestroyWindow Test1
```

上面的输出意味着干净的资源管理

```
------------------------------

OnCreateWindow Test1

OnCreateWindow Test1

OnDestroyWindow Test1

OnDestroyWindow Test1
```

上面的输出可能意味着底层使用了引用计数进行管理

```
------------------------------

OnCreateWindow Test1

OnCreateWindow Test2
```

OnDestroyWindow Test2

OnDestroyWindow Test1

上面的输出可能意味着底层使用栈的方式管理窗口，先进后出。

无论怎么样，在对称的艺术下，我们就可以去更多的关注设计层面。我们来看一个例子。

在某个凌晨的夜晚，睡在梦中的老A被某个电话叫醒。他收到来自小B的一个截图。截图上面只有一小段，

OnGateConnect 1 0

OnGateConnect 2 1

OnGateDiscconnect 2

OnWorldDiscconnect

表现的现象是无法参与跨服玩法了。根据经验，已经可以猜测OnWorldDiscconnect就是导致跨服无法参与的原因。那么导致这个跨服失败的情况可能是什么？小B的分析是即使上面出现了AAB型日志，但是因为它不是一个完整的生命周期（进程启动到结束是一个生命周期，这时候进程还在运行，生命周期并没有结束），所以这个输出不能得出什么有效信息。老A在床上看着这个输出，陷入了思考。

是的，这不是个完整的生命周期，所以是会出现AAB型的日志的。但是假如OnGateConnect 1是正常的连接并且没有触发OnGateDiscconnect，为什么会出现OnGateConnect 2与OnGateDiscconnect2的对称，部分对称部分不对称是不合理的。依据这个不合理，老A最终得出了OnGateConnect 2从出现就是不合理的结论，避免了一次通宵。

所以，在非生命周期中，对称也可能变成一种异常。而在生命周期内，非对称往往是一种异常。这种非生命周期对称异常和生命周期非对称异常都值得我们关注。

6. 代码修改与重构

在漫长的开发周期中，小C遇到了一些需求。当他准备开始撸相关的代码的时候，他发现隔壁的小B之前已经实现好了类似的功能，于是他开始纠结要自己写还是复制黏贴。思来想去，他决定先看看能不能拿过来用吧。

小C研究了下，总结出使用他人的代码有几种方式：

1. 别人写了一个class test，我们在自己的类内部对它进行组合。

2. 别人写了一个class test，我们继承它使用。

3. 别人有个函数，我想复制它的部分代码。

4. 别人有个函数，我想直接调用，但是参数可能比较多，也可能比较少。

那下面就要探讨这几种使用方式。首先假定我们需要使用到的功能大部分就是一个对象已经实现好的功能。那么这时候我们可以考虑选用前面2种方式。对于第2的方式，很直观的可以知道如果是继承，那么父类对子类的影响很大，因为父类可能会在自己的类中加入不确定的成员，另外如果我们按照这种思路，一个子类将继承自多个父类。多重继承带来的耦合性是非常大的，也是很多语言不支持的。那么如果我们使用方式1呢，耦合性会变低。在使用层面上，如果需要对外暴露引用到的对象的相关接口，会没有继承那么方便，有可能需要重新套上一层接口，也有可能需要对外暴露引用的类对象。

```
class Test{

    public addRes();

    public delRes();

}

class C extends Test{

}//外界可以直接调用addRes

class D{

    private test:Test = new Test;

    public addRes(){

        return test.addRes(); //封装的方式
```

```
        }

        getTest(){

            return test; //提供对象接口的方式

        }

    }
```

下面我们来重点讨论一下如果我们需要去使用别人函数的情况。如果我们只是需要里面的一小段代码，并且我们预估大概率用不到那个函数里面其他的功能，那么这个时候我们把那个目标函数复制一份，进行修改。如果我们无法做出预估，那么我们需要在那个函数上面进行改动。如果我们需要加参数，那么我们可以使用前面说过的tOption方案。如果我们需要的参数没那么多，比如原来的函数是5个参数的，且都不能为空。但是我们只用到2个参数，比如下面的情况：

```
function getPath(nStart, nEnd, nType1, nType2, nType3)
```

我们只需要nStart, nEnd，那么我们应该怎么办？小C不喜欢遇到这种情况，他想简单的给nType1, nType2, nType3赋值为0，然后去调用这个接口，我们假设项目里面大部分number的隐性默认值即为0。这个做法有点不舒服，于是小C进到那个函数里面看了一下，传0确实是ok的。但是这样一来又有点担心，万一以后加参数了怎么办？

这边其实有个更重要的思考，就是我们可以重构这部分代码，使其更合理。当我们的参数个数少于要调用的函数的参数个数时，我们要尝试把多余的参数剥离出来，让原来的函数调用我们新写的函数。最后期待的改造是这样子的：

```
function getPath(nStart, nEnd, nType1, nType2, nType3){

    switch(nType1){

        //

    }

    getPathInner(nStart, nEnd);

}

function getPathInner(nStart, nEnd){

}
```

遇到使用他人代码的时候，边重构边使用是更加有利于项目的一个方式，之前大部分程序的观点是尽量不要去改别人的代码，这种观点是值得商榷的，因为它可能会使得原来不舒服的接口造成更大的影响。我们在项目中秉持一个原则，如果有个接口让你不舒服，比如说多传了几个参数，那么我们一定要提出来，那一定是接口的设计不够简单或者没有提供更简单的接口形式。再深入讨论一下，如果你想要的接口的参数与他人提供的差距很大，这时候应该怎么办？

```
function addItem(nType, nCount, …)

function addItem(sUniqueId, nCount, …)
```

当出现这种情况时，我们可以允许存在2种类似的接口，但是，我们一定需要把它们的共同部分抽象出来，作为一个单独的函数。不断抽象也是在使用他人函数的时候要做的一个重构，也能使项目变的更好。

最后，简单总结下。使用他人的代码不是简单的复制黏贴，如果他人写的够好，那么就直接引用。如果是写的不够好，那么我们就去重构。只有互相重构，才能使得整体代码效率变高。

7. 优雅的使用外部代码

在前面的章节中，我们谈到了使用项目中他人写的函数，我们采用的是一边改一边重构的方式，使之更符合整个项目。那我们使用引擎层的代码有什么注意的吗？

使用引擎层的代码，我们要尽量使用局部缓存缓存住接口或者值。举个例子：

```
let levelSlider = Core.createBitmapByName("slider_png");

levelSlider.x = 0;

if(XXX){

    levelSlider.x += 6;

}

if(XXX){

    levelSlider.x += 8;

}

if(XXX){

    levelSlider.x -= 2;

}
```

上面的代码里面对levelSlider.x 这种操作，我们没办法直接知道这个赋值行为也就是levelSlider.x= 底层是不是做了很多操作。所以我们需要先把值缓存住，等计算完成后再一次赋值过去。

修改后的代码为：

```
let levelSlider = Core.createBitmapByName("slider_png");

let nSliderX = 0;

if(XXX){

    nSliderX += 6;

}
```

```
if(XXX){
    nSliderX += 8;
}
if(XXX){
    nSliderX -= 2;
}
levelSlider.x -= nSliderX;
```

对于引擎代码，我们应该尽量保持警惕，尤其是在每帧都执行的函数中，很容易产生大量的gc或者是消耗大量的CPU。我们的一个原则就是要尽可能少去直接和引擎进行交互，而是更多的进行局部缓存，把战场拉回到更加通用的逻辑里面。对值的设置，或者是获取都是可以进行这样的优化。

我们再来看一个例子：

我们想对一个窗口内的一个ui调整层级。我们有2个方式可以完成：

1. this.addChildAt(oImgBottom, nIndex);

2. this. setChildIndex (oImgBottom, nIndex);

1,2 两个方式都可以实现调整ui层级的作用，但是我们推荐是使用第二种，因为第二种的写法字面表明的含义在这个地方更加符合我们想要做的事情。第一种写法是重新添加对象，第二种写法是设置对象的索引层级。很明显第二个写法更加清晰。很多情况下，我们在使用第三方引擎的时候，比如unity、egret、laya等引擎都会遇到一些超乎预料的未知bug，这个过程常常被程序员称为踩坑过程。有些同事会经常遇到，有些同事就完全不会遇到。我们尽可能的使用容易理解，清晰的接口，这样我们就可以尽量避开引擎本身的各种坑。

再看一个例子：在某个引擎的粒子系统内， 有2个成员变量：

emitterX 对应的注释：表示粒子出现点X坐标

emitterXVariance 对应的注释：表示粒子初始坐标 x 差值

两个变量貌似都可以影响粒子出现点的坐标，那我们在第一次写代码的时候应该选用字面意思更为容易理解的变量，也就是emitterX。不需要也没必要去搞清楚这2个变量的

区别，在需求不满足或者与预期不符的时候再去弄懂即可。这可以节约我们在引擎使用上面的不必要时间成本。

前面我们讲了我们在使用底层代码的时候应该尽量使用简单的部分，也尽可能少的调用它们来实现自己的目的。当底层代码提供了不同的接口或者参数来做类似的事情的时候，我们也应该尽量选取里面简单的部分。在白鹭引擎里面，有个接口是这样的：

```
hitTestPoint(x: number, y: number, shapeFlag?: boolean): boolean;
```

这个接口它提供了一个shapeFlag参数用于指定是使用像素检测或者是边框检测。在我们的项目中，也许传递不同的参数不会有什么影响，但是我们应该优先选择边框检测。选择边框检测的原因是因为边框检测的复杂度比像素检测低，像素检测依赖于更加底层的接口，而边框检测是个通用的逻辑技术，也底层无关。曾经有个白鹭项目里面使用像素检测，后面跑在微信小游戏上的该项目突然就无法正常工作了。微信的bug影响了我们上层的逻辑bug，这也告诫了我们应该优先使用简单的接口。

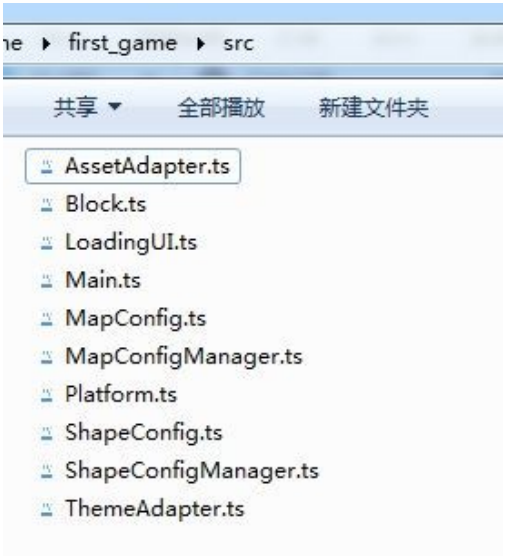
选择简单的接口/参数以及尽量少的使用/调用底层接口就是我们所谓的正确的代码使用方式。

8. 代码审查

代码审查是通过直接阅读代码的方式来发现代码的问题以及优化代码，通常我们也称为code review。代码审查是主程序的必修课，它可以有效规避代码风险以及建立团队的代码通识。有些项目会采用团队成员互相进行代码审查的方式来共同成长与提高。

我们来实例看一个 code review 的例子，使用白鹭引擎开发的一个拼图游戏：

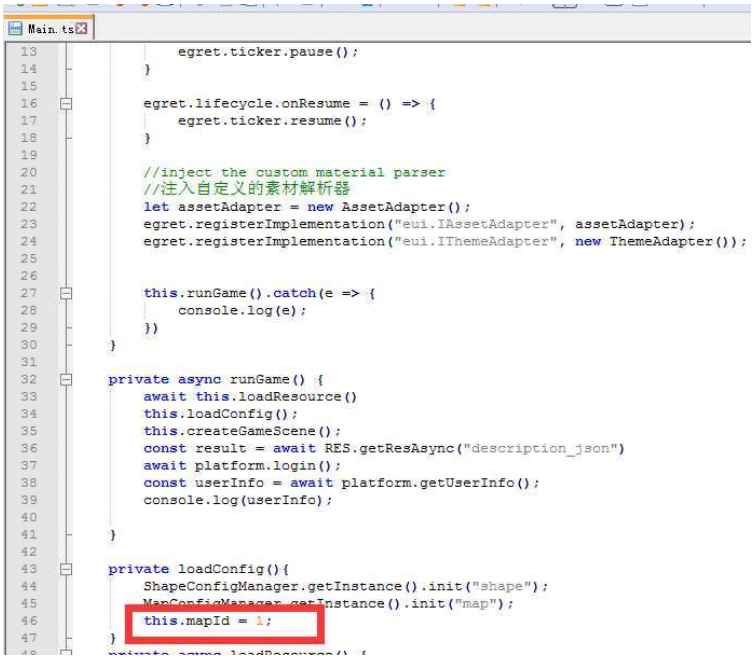
我们第一步先不着急看代码内容，先看文件的规划（如下图所示）。下面是一个小游戏的代码文件，它实现的目标是一个拼图逻辑。



从名字我们可以看出，这里面好几个文件都是做配置的支持的。这个文件的规划有3个问题：

1. 一眼看不到核心实现文件。
2. 配置的支持应该放到一个单独的文件夹。
3. 配置的支持应该尝试用一个类来实现。

接着我们看内容，我们先来看一下Main文件，Main文件是入口文件（如下图所示）。



看到一个奇怪的地方，入口文件里面定义了一个逻辑变量mapId。入口文件应该干的事是一些比较重要的，在项目早期就必须提前执行的逻辑，它不应该和游戏的核心逻辑产生任何的关联，仅仅应该是在适当的时候将控制权交给游戏逻辑层。

```
private textfield: egret.TextField;
private play_btn: eui.Button;
private blockIntervalX = 60;
private blockHeight = 900;
private touchBlockBeginX: number;
private touchBlockBeginY: number;
private mapId: number;
private blockMap:{[blockId:number]:UI.Block} = {}
```

接着往下看，到了变量命名的地方（如下图所示）：

这边的命名是很糟糕的。对于2个不同单词的分割，有些直接拼合，有些用下划线分割，有些用大写分割，没有做到统一。统一是非常重要的，就像军队的纪律一样。另外一个我们应该需要注意的是，不是object的对象类型，也就是那些基础类型，如boolean，number等我们应该赋予它一个初值。上面的mapId我们应该设置一个0，防止对一个空对象尝试误用。

```
protected createGameScene(): void {
    let bg_black = new egret.Shape();
    bg_black.graphics.beginFill(0x000000);
    bg_black.graphics.drawRect(0,0,this.width,this.height);
    bg_black.graphics.endFill();
    this.addChild(bg_black);
```

接着我们看一个函数的实现(如下图所示），这个函数的实现过度复杂了。

这段代码很好读懂，就是创建了一个黑色的背景。创建一个黑色的背景从直观的理解上，应该是一个简单的操作，那么它如果需要写5行代码，那么需要怀疑是不是实现复杂了。另外一方面，有些程序会用一张黑色的背景图来拉伸实现这个效果，也可能实现过度了。我们在unity里面经常为了防止点击到窗口后面的东西，我们会加一个透明的背景来接收鼠标事件，这也可能是实现过度了。实现过度指的是使用相对消耗的方式来处理一个简单的逻辑。

接着我们再看一个函数的实现（如下图所示）。函数的名称不重要，我们关注函数的

```
private setOtherBlockTouchInvalid(validBlockId:number):void{
    for(let k in this.blockMap){
        if(this.blockMap[k].m_id != validBlockId){
            this.blockMap[k].touchEnabled = false;
        }
    }
}
```

内容。

在脚本语言中，从this上面去取数据是一次的查表操作，查表操作对于脚本语言而言是存在一定耗时的，尤其当数量变多了之后。所以上面的this.blockMap应该先用一个局部变量缓存住，以减少查找的次数。另外一个this.blockMap[k]，因为它的使用次数介于1-2之间，所以它不一定需要用局部变量缓存。局部变量缓存加快的速度，但也可能造成gc的负担。本质上它是一个空间换时间的策略，而在游戏里面，CPU相对内存一般是CPU先达到瓶颈，所以我们经常会优先关注时间的实现效率。

接着我们看到一个函数的内部，进行了取配置的操作（如下图所示）。

```
}

private displayBlock(mapId:number): void{
    let mapConfig =  MapConfigManager.getInstance().m_mapInfoMap[mapId];
    let shapeIdInfo = mapConfig.blockIdInfo;
    for(let i = 0; i < shapeIdInfo.length; i++){
        let shapeConfig = ShapeConfigManager.getInstance().shapeConfigMap[sha
        let block = new UI.Block(shapeConfig.shapeId,shapeConfig.shapeInfo,sh
        this.blockMap[shapeConfig.shapeId] = block;
        this.initBlockEventListener(block);
        this.addChild(block);
    }
}

private displayMap(mapId:number):void{
```

这里，先取出一个管理器对象。然后直接访问了它的成员函数，这是非常不应该行为。第一个是这个成员不应该是公有的，而应该是私有的，它的公有性质破坏了类的封装。我们应该用提供接口的方式来替代这一种写法。这种写法最大的后果是当它出现异常的时候，我们无法通过单一的接口来排查它的修改来源。切记我们不应该直接访问m_或者$开头的私有变量，除非迫不得已。迫不得已打个比方说它是被引擎包在了内部不好，这种就不好直接修改引擎提供接口，就只能直接去访问了。

我们写很多逻辑的时候，常常可能使用过于复杂的方式去实现一个简单的逻辑，下面

```
}

private onTouchBegin(e: egret.TouchEvent){
    this.touchBlockBeginX = e.stageX;
    this.touchBlockBeginY = e.stageY;
    le block: UI.Block = <UI.Block>e.target;
    this.setOtherBlockTouchInvalid(block.m_id);
}
```

这个图就是这样的一个案例。

这个地方有个明显的逻辑设计问题。我们从名字上看，这个过程是在鼠标选中某个物体的，使其他物体的点击响应失效。我们应该建立一个认知，我们在处理一个物体的时候应该尽可能不要去主动操作别的同级物体，注意这边的同级2字。那么我们有一些别的方式来实现，比如说在onTouchBegin的时候把它置为同级物体的最上层，这样事件就会被它先截获，也不用去关闭其他所有物品的事件响应了。

下面一个是代码实现的效率问题，在for循环里面（见下图）。

```
for(let i = 0; i < map_dot_info.length; i++){
    if(map_dot_info[i][0] - interval < block.m_leftUp_x && block.m_leftUp_x < map_dot_info[i][0] + interval &&
       map_dot_info[i][1] - interval < block.m_leftUp_y && block.m_leftUp_y < map_dot_info[i][1] + interval){
        offsetX = map_dot_info[i][0] - block.m_leftUp_x;
        offsetY = map_dot_info[i][1] - block.m_leftUp_y;
        //console.log("ycj x,y:",block.m_leftUp_x,block.m_leftUp_y,offsetX,offsetY);
    }
}
```

for循环里面常见的漏掉break。虽然代码也能正常工作，但是效率不够高。

上面我们看的是一个游戏职场新入门的程序写的代码，下面我们来review一点引擎代码。以白鹭中的一段引擎代码来举例，我们来看看如何去审查它的问题。

```
Timer.prototype.$update = function (timeStamp) {

    var deltaTime = timeStamp - this.lastTimeStamp;

    if (deltaTime >= this._delay) {

        this.lastCount = this.updateInterval;

    }

    else {

        this.lastCount -= 1000;

        if (this.lastCount > 0) {

            return false;

    }

    this.lastCount += this.updateInterval;

    }

    this.lastTimeStamp = timeStamp;

    this._currentCount++;

    var complete = (this.repeatCount > 0 && this._currentCount >= this.repeatCount);

    if (this.repeatCount == 0 || this._currentCount <= this.repeatCount) {

        egret.TimerEvent.dispatchTimerEvent(this, egret.TimerEvent.TIMER);
```

```
        }
        if (complete) {
            this.stop();
            egret.TimerEvent.dispatchTimerEvent(this,egret.TimerEvent.TIMER_COMPLETE);
        }
        return false;
    };
```

上面展示的是白鹭引擎 Timer的update函数，我们来看看里面有哪些值得写好的地方。

1. 变量使用2种命名方式 lastTimeStamp 与 currentCount。

2. this.lastCount -= 1000; 硬编码的1000，无法快速读懂含义。

3. var complete = (this.repeatCount > 0 && this._currentCount >=this.repeatCount)； 与使用 complete的地方没有挨着一起（中间隔着处理的别的事情，影响代码阅读）。

4._delay看起来是延迟相关，配合return false来实现延迟处理一个事情，实现方式读起来很困难。

5. 多次使用this.lastCount，没有进行合理的引用，也就是

```
var lastCount = this.lastCount;
```

我们的code review就暂时进行到这里了。可以看到，一份代码很容易出现各种各样的问题。这些问题有大有小，之所以我们都拿出来说的原因是这些代码经常是被复制黏贴，导致一个不够好的代码出现在各个地方。当然，最重要的是我们要培养自己的条件反射意识。比如说创建了对象就要关注销毁，遇到for循环就要找找是否有地方可以提早break。不停的培养代码意识是提高代码的有效方法。如果是主程或者高级工程师，那么可以开始看看项目中的其他成员的代码来训练code review的能力。

9. 从面试的角度看面试

我们这一节会看到一个简历里面的部分内容，从这里面我们来看如何面试以及如何应对面试。

我们在应届生简历中会看到一个技能栏目：

1. 熟悉javascript
2. 熟悉白鹭
3. 熟悉photoshop
4. 熟悉html

其中的第三点是不该写入的，因为它不是一个程序技能，写进去反而是代表了对游戏技能认识的不足。另外白鹭引擎中最有开发效率的是typescript语言，在上面中列了在工程项目中基本不会用到的js以及html，这表现出来的也是对技能认识的不足。

接着来看实习经历：

接触了P2物理引擎以及粒子系统，了解json等数据结构，各种精确碰撞，矢量绘图。学习了TextureMerge并且使用它进行合图切图。熟练使用工厂模式进行游戏开发，也遇到和很多没有遇到过的问题。

上面的内容从一个面试官的角度可以这样归纳：

1. 了解物理引擎与相关碰撞
2. 了解白鹭相关工具与使用
3. 了解一些设计模式
4. 整个实习经历能提高能力

那么从上面的几点，我们可以提一些问题来考察他的能力。

1. 如果不使用物理引擎，怎么判断2个圆相碰撞？
2. 白鹭有哪些工具提供给开发者使用？
3. 合图有什么优点，有什么缺点？
4. 你还了解哪些设计模式？

5. 工厂模式应该在什么时候使用？

6. 实习过程遇到问题是怎么调试的？

7. 如果有个按钮点击了没响应，如何快速找到问题？

结合之前的技能栏目，还可以从语言，引擎层面延伸一些问题：

1. typescript 与 javaScript的区别是什么？

2. typescript中用的最爽的是什么，最不爽的是什么？

3. exml是什么？

4. [] 与 {} 有什么区别，分别什么时候使用？

5. 数组与字典的查找速度哪个更快，为什么？

上面的部分主要是针对简历问。当我们在面试一个新人的时候，更多的是看潜力和态度。如果要看他的潜力，我们需要准备更多的问题去尝试了解面试者对技术的热爱以及对知识的热爱。面试潜力的方法如下：

1. 提问

2. 对面试者的回答进行分析，补充。以补充后的信息继续提问。注意这边的问题应该是面试者通过当前信息进行思考能够回答或者部分回答的。

3. 针对每个问题，给足时间。重复2的操作。

4. 观察他对问题的思考角度以及思考耐心。

5. 在最后离开前，让他再次思考对项目以及团队，公司的期待。

通过上面的不断的要求面试者思考，思考，再思考。我们可以尽可能得出面试者对问题的处理态度，这个态度决定了他是否能在将来遇到问题的时候自我成长。

除了面试初级的工程师，我们还会面试中级，高级工程师。中高级工程师的面试分为2个部分：

1. 基础能力。当看到简历上面写着熟悉网络编程，我们就可以深入的问TCP的握手以及断开连接的过程。比如问TCP需要几次握手，为什么是3次？

2. 逻辑能力。大部分工程师去到公司还是写逻辑的，所以逻辑部分也要重点问。比如说如何实现一个关卡系统，如何跨服，如何分线。

这些内容也是本书的主要内容。

3. 硬核能力。硬核能力大部分是上线的问题，以及一些比较难处理的问题。甚至是一些语言交互，热更新等问题。这些问题往往伴随着很多限制。比如说线上服务器不响应网络请求了，如何排查？

作为面试者，遇到不懂的问题多思考就好。确实不懂也没关系，因为很正常。如果换位面试，对方也是很多不懂的。因为有限的时间内，大部分人的技术可达范围都是有限的。要获得超出一般人对代码的认知，多半是通宵熬夜熬出来的。

10. 如何应对代码错误

在我们日常写代码的过程中，即使有IDE的辅助，我们依然可能会犯一些错误。这些错误超出了IDE的解决范围，可能需要写代码的人非常的小心与注意。对于新手程序，细心往往是个需要长期培养的事。对于这种情况，我们应该如何避免类似的错误发生？

我们来看一个例子：

```
egret.Tween.get(this.imgPlayer,{loop:true}).to({ alpha: 0};
```

上面的例子中，想对imgPlayer进行一个透明度变换。但是一直没有生效，真实原因是这边的this.imgPlayer实际上不是一个成员变量，而是一个函数，所以这边的正确做法是：

```
egret.Tween.get(this.imgPlayer(),{loop:true}).to({ alpha: 0};
```

对于我们这个例子，最终的修复手段不是那么的重要。我们要思考的是如何规避这种错误。

对于这个问题，我们应该认识到egret.Tween.get并没有对传入的对象做检测，对于这个库而言，这个检测不一定是它的责任，理论上它允许对一个函数进行变换也不是什么大的问题。但是，对于我们的逻辑，我们基本不可能对一个函数进行变换，所以在我们的项目中，就应该具备这个检测。所以我们可以将这个函数进行封装，并要求大家统一都使用封装好的函数。在这个封装好的函数内部，对第一个参数进行判断，如果是函数就直接断言。

封装是我们应对常见的项目问题的一个有效的处理手段。在后面的例子中我们将会更多的看到类似的处理手段。当然，封装仅仅只是一个手段，我们还可以从工具链的角度来规避这些可能的犯错。

这边我们还可以认识到统一规范的重要性。当项目中存在了一个实现与底层接口一致的时候，我们应该优先选择项目中的实现。对于一个程序而言，他应该更多的时候大家封装好的接口。对于一个策划而言，他只能通过配置生成为程序需要的配置，而不能直接去手动修改最终生成的配置。统一化的流程意味着我们可以在各个阶段对错误进行有效的规避。

面对错误，除了代码的手段来规避外，作为项目的主程，还需要统一思想。这个过程包括要求大家遵循统一的代码命名等，这也是规避错误的一个重要手段。越是相似的代码风格，代码的阅读速度就会越快。

第二章

逻辑的设计模式讨论

11. 分层设计

在我们项目开发中，下到底层核心代码，上到逻辑层应用，都离不开分层思想。

分层思想旨在将不同的依赖关系分离开来，并减少代码的耦合度，使得代码的维护更加方便，也更加合理。我们先讨论一个分层框架或者说写法，MVC。在很多地方，都会使用到mvc模式，尤其是web中。我们在这边讨论游戏中是否应该使用MVC。如果是在游戏中使用，那么我们会怎么写一个ui视图的代码呢？

M->负责数据

V->负责显示

C->负责响应

看起来是可行的，很清晰。那么它对比MV会怎么样？

1. 需要建立3个文件，在control层面没办法很快的找到V里面对应的定义(如果编辑器不支持快速跳转的话)。

2. 游戏的V（视图）往往只有一种呈现，没有变化的V，VC合并在一起更有利于开发速度。

在知乎上面也有一个文章讨论游戏是否需要MVC架构：

https：//zhuanlan.zhihu.com/p/38280972

那么我们在游戏中是否需要使用MVC模式，老A的结论是：用也不用。

用在哪里：在网络的消息往来中，收到一条信息，我们需要修改本地的模块数据，我们需要更新界面，那么网络消息处理的handle模块，扮演的就是C，C的作用就是更新M和V。

不用在哪里：不用在ui界面的逻辑中，因为ui界面经常整个换或者调整，它需要快速的开发效率。把单个ui的逻辑写在自己的一个view文件中，在这个文件中使用M，这样是最快的方式。

MVC带给我们的更多是分层的思考，那么我们就要聊聊分层的一些原则。

第一个原则：在分层设计中，事件的派发遵从从下往上。

什么意思？越上层的逻辑通过监听下层的事件来实现自己的需求。比如我们说ui逻辑上的事件点击是上层，引擎提供的事件派发就是下层。那么一个界面v和数据层m之间是什么关系呢？v经常使用到m，所以我们把m定义为v的下层，也就是m更加底层。v通过监听m层的事件来实现自身的变更。比如说v里面有个等级level，在等级变更的时候m层会发出

事件，监听它的v就会做出响应。下面我们再讨论几个场景，来试图理清楚我们到底是如何分层以及层与层之间到底有什么关系。

第一个场景，在一个游戏里面， 主界面的更新与玩家的移动是什么关系。

我们经常看到的写法是：主界面有个摇杆逻辑，在摇杆逻辑中触发了玩家的移动。摇杆逻辑触发事件，玩家响应。或者一般大家就直接调用玩家类的移动方法。而在玩家类中，当有玩家的等级发生变更，这时候主界面的等级又需要改变了。可能很多人就在玩家类里面就直接调用主界面的接口去修改主界面的表现了。

首先我们说一下这样调用存在的一些问题。主界面和玩家数据因为异步的关系，它们很可能是其中一者优先初始化好的。当一方需要调用另外一方的接口的时候，很可能需要去判断它的就绪状态，这是很烦人的事情。正如第一章节中的配置一样，如果我们每次使用配置前都需要去判断一下配置是否存在，那么我们的代码将掺杂着很多判断代码，也就不容易读懂了。在小C的成长过程中，他就是这么写代码的。他的想法是：在某些情况下，高的耦合意味着更好的效率和逻辑的实现。只要这个耦合能控制在一定范围内，这个耦合是可以适当被允许的。

那么这个真的是正确的设计吗？我们常常听到的词是高内聚，低耦合。与这个建议似乎是相违背的。在这个场景下，玩家类和主界面的逻辑就不该掺和在一起。但是我们明明看到玩家类依赖于摇杆事件，主界面依赖于玩家数据，他们看似就该是一体才能比较互相依赖。

首先大部分玩家类是含有玩家数据的，我们按照上面的mv的层次划分，我们简单的把他们拆开。因为m是数据本身，v是视图，视图是经常变的，我们暂时认为m更加底层，一会我们会更加详细的说明这个认识的本身。那如果这样划分，摇杆怎么办？我们把摇杆再拆出一层，这边变成了3层，摇杆，主界面，玩家数据。我们把顺序从上到下理一下，主界面，玩家数据，摇杆。

通过拆出额外的一层，解开了主界面与玩家数据的耦合。摇杆的响应是可以作为一个单独的模块移动处理的。主界面根据不同的摇杆状态显示不同的界面表现，玩家根据不同的摇杆状态进行移动，释放了玩家和主界面。我们还有另外一个判断摇杆是否可以独立一层的逻辑，想象你把摇杆逻辑平移到了另外项目，是否需要拖家带口把其他文件复制过去，如果不需要，那么恭喜你，你的摇杆逻辑的分层是对的，且逻辑是内聚的。在上面的案例中，也牵扯出了我们的另一个原则。

第二个原则：层的上下层次的划分根据依赖性进行划分。越重要的，越被依赖的层位于越下面。玩家移动依赖于摇杆，那么这个依赖导致了摇杆更加的底层。

而玩家数据经常会被主界面或者其他界面进行依赖，所以它位于界面层的下面。这边我们可能会产生一些问题，就是摇杆的层次是否应该在玩家类的下面。我们还可以再拆，把玩家类中的移动和玩家数据拆分开。那么这时候，玩家类的移动（表现为界面人物的移

动）和玩家的数据（等级，vip等）是分开的。这边的层次关系就更加清晰了，人物表现变成了单独一个层，下面是摇杆层。玩家数据层在ui层（主界面等）的下面。玩家数据和摇杆层之间没有任何关系，它们是独立的关系，存在在独立的分支上面。当有一天它们需要依赖的时候，我们才会给他们的层次做个简单的排序。

第二个场景，在玩家的数据中，存在3个模块。基础属性模块（含等级，属性等），vip模块以及战场玩法模块。其中，

vip模块以及战场模块依赖于基础属性的等级进行开启

基础属性中的经验获取依赖于vip的种类与等级

vip模块与战场模块都会对属性模块中的属性进行加成，属性模块的属性是算所有模块的总属性

这样的几个关系似乎也导致了它们之间的强依赖。小C对它们进行的层级的排序，认为它们互相依赖，应该是平级。有点工作经验的小B不这么认为，它觉得从直观上就是战场模块在最上面，vip在中间，基础属性模块在最下面，毕竟都叫基础模块了。那么我们首先看看战场模块是不是应该在最上面。在前面的例子中，基础属性模块的属性计算依赖于战场模块。我们考虑将属性相关的接口抽象成一个AddAttr（加属性）以及ClearAttr（清除属性）接口，需要的模块自行调用这2个接口进行属性的添加与清除。这么一来，基础属性模块就不依赖于战场模块了。

接下来的部分会比较难拆了，基础模块和vip模块都相互依赖于对方。vip模块依赖于等级的变更，而基础模块依赖于vip种类与等级，是依赖一个状态。依赖等级可以用事件的方式解耦，依赖状态可以使用跟战场一样的方式进行解耦。所以无论我们是依赖一个状态还是一个状态的变更，我们都可以用一些手段进行解耦。这就得出了我们对分层的另外一个认知：层与层之间是可以独立存在的，不一定需要建立层次顺序。

那么我们前面说到的分层顺序是否还有意义？层与层之间互相独立是否是最佳的设计？如果你按照层与层完全独立的写法写代码，代码之间的耦合度是非常低的，但是写的过程会发现，并没有那么有效率。为什么，因为你没办法随便用别的模块的一些接口了。如果你的经验倍率依赖于vip等级，那么你提供的是经验添加倍率接口AddExpFactor，在你的经验计算中，使用这个倍率进行计算。而vip模块调用添加倍率接口影响倍率。它们之间只依赖于行为，而不依赖于具体的某个属性。本来可以直接GetVipLevel获取vip等级的，不能用，只能设计一个机制给其他模块调用。

那么进化的设计就是：如果某个属性，比如经验只依赖于vip等级，那么我们还是想直接调用，为了效率。而当出现多个依赖时，我们再将原来的依赖改为某种机制，解除依赖。但是，如果依赖的是战场模块里面的某个属性，我们就不能直接去调用了。因为战场模块是玩法模块，是高度变化的。今天代码没写完，明天功能可能就改了。所以对于它的依赖，只能是通过建立机制来解耦。

第三个原则：越容易变化的层位于越上面。就像战场模块这样的玩法层面，变化很大。而我们把基础属性模块，vip模块，邮件模块等相对稳定的模块放在下面。这种划分有利于我们把易变的和不易变的划分开。

这3个原则不是一定要强制遵循的，而是为应对游戏行业行业而生。如果没有一定的限制条件，我们依然是可以保持层与层之间的独立。那么这些限制有：

1. 开发节奏快，程序经常需要加班，开发效率成为开发过程中不可忽略的事情。

2. 需求的变动是频繁的，今天在开发的系统，明天可能就要推倒重来了。

3. 即使是大面积的改动，也要保证后续的开发效率

所以代码既要耦合低，也要速度快。于是老A定了一个新的项目代码分层规范：基础属性模块，vip，邮件，任务等模块变更较少且经常被其他模块依赖的，作为基础层。其他逻辑模块作为逻辑层。

基础层之间的模块允许自由的使用对方的接口，或者依赖对方的接口。

基础层位于逻辑层之下，只能给逻辑层派发事件或者提供行为接口而不允许依赖于任何的逻辑层状态。

逻辑层之间也不允许互相依赖，只能保持独立。

这样的分层保证了开发效率以及代码的可维护性。同时，因为逻辑层的低耦合，我们很容易的将一个不需要的模块删除，而不用担心其他地方依赖到它导致脚本崩溃。

再聊一聊层与层之间的关系，还是拿主界面和一个玩家角色来说明，我们这边假设这里的玩家角色只是一个显示对象，主界面的更新部分依赖于玩家角色的在场景里面的行为。那么对于这样的2个界面，我们可以把它们归属成不同的层。比如说玩家角色位于场景的显示层（里面可能有地图建筑等），主界面等ui一般出现的时候都是显示在场景上面的，它们单独为ui层。下面我们来看2种对主界面和玩家角色的关系理解：

1. 玩家角色位于主界面对象的里面，主界面对象拥有一个玩家角色对象。

2. 玩家角色对象和主界面对象完全独立，由更高一层的逻辑，比如场景逻辑负责创建它们，并建立消息通知机制。

根据我们上面的一些结论，我们很自然的可以使用低耦合的方式，也就是b的方式来建立它们的关系。但是a的关系设计也是会经常看到的，就是在小游戏中。因为在小游戏中主界面和玩家角色的创建都是非常快的，也就是说少了很多异步创建等情况。这使得我们可以使用a的方式来建立关系。总结下，当我们满足以下情况时：

1. 并发异步创建的情况少

2. 创建的速度足够快，基本是阻塞的

3. 项目规模小，预判未来的需求变更是小的

我们可以采用更加灵活的层关系来表达一个游戏内的依赖关系，因为分层的优势在于代码清晰，很好的应对需求变更。而在足够小的游戏方面，这些优点无法完全的发挥效用，所以我们可以采用其他的层关系。注意了，当一个小游戏的核心不变，但是在不同的环境（h5，微信小游戏）有不同的表现的时候，我们依然要用回分层思想来应对不同的环境。

12. 主动与被动

我们本小结讨论一下主动与被动的设计。先举个例子来描述什么是主动，什么是被动。

我们需要设计一个引导系统，当出现某个窗口的时候，引导系统将作出一些反应。那么这边我们有2种方式实现这个需求：

1. 当有个窗口打开的时候，通知机制层向外面发送一个窗口打开事件，引导系统监听这个事件并作出响应，这是被动模式。

2. 引导事件定时去查询监听的窗口的状态，如果发现了窗口的状态是需要的状态，那么就作出响应，这是主动模式。

那么我们比较一下主动和被动的一些优缺点：

主动模式因为需要轮询状态，与被动方式相比，没有那么的及时。很多情况下，这种轮询都没有触发事件。如果轮训的状态需要通过一定的计算，那么这种方式是很浪费CPU资源的。

被动模式的事件发生的地方是在触发的那一刻，事件的触发和响应是连在一起的，这给调试带来了极大的便利，可以很容易的找到为什么没有触发对应的行为。而主动模式的响应是滞后的，意味着它的响应实际上比目标事件发生的时间晚。具体晚多少取决于主动轮询的频繁程度。

主动模式有利于实现多条件类型的监控，比如它可以监视多个状态的改变。而被动模式虽然也可以监听多个条件，但是在条件的变更中，被动模式很不灵活。被动模式监听的通常是触发这样的行为，而当一个状态没有发生变更的时候它就显得比较乏力。

举个例子：引导系统需要在窗口A关闭的时候，且窗口B打开的时候触发一个行为。

对于主动模式而言，它只需要轮询窗口A和窗口B的状态即可。而对于被动模式，它可以监听窗口B的打开事件。但是窗口A从头到尾都没有任何状态的变更，监听的也不是窗口A的关闭，这时候就比较尴尬了。

在主动模式的间隔内，可能会遗漏掉变更。比如主动模式是1秒判断一次，那么如果玩家在1秒内完成了打开窗口到关闭窗口这样的操作。主动模式可能就没有办法正常的触发了相关的行为了。当然，这个问题可以有不少解决方案。比如修改判断间隔，或者忽略这个情况，因为这个情况事实上也不会造成什么体验的问题，只是在逻辑上存在这样的一种情况。

这2种模式各有有缺点，通常在设计之初，会使用被动模式，方便调试。当后面被动模式无法支撑条件的时候，或者条件存在与或取反等结合的时候，我们需要加入主动模式。一个系统如果是设计成主动被动可自由切换或者可以一起使用的话，这将是最理想的。

这边我们还需要再更深的讨论下这2种模式。我们说主动模式在某些情况下其实是有一定效率优势的，当事件触发了多次时，主动模式只会在后面去取状态进行判断，而被动模式会多次响应执行。另外，很重要的一点，主动模式没有立即响应，也就避免了死循环以及重入。举个例子，在一个以被动模式为基础设计的关卡系统里面，在它的某个关卡的条件里面，策划配置了怪物数量小于0触发通关。怪物死亡的逻辑是死亡，触发事件，从管理器移除。 当最后一只怪物死亡的时候（怪物数量变更）触发了关卡通关，关卡通关清理场景怪物又导致了怪物的死亡，于是出现了死循环。我们需要有经验的被动模式设计者才能提前做好准备，比如在怪物死亡的时候先判断一下是否已死。如果是主动模式的话，那么怪物死亡的时候不会触发任何东西，在下次轮询的时候，关卡获取到当前的怪物数量触发通关清理怪物。这时候之前的怪物已经正常从管理器里面移除了，也就不会被多次销毁。

13. 阻塞与非阻塞

首先，我们这边所说的阻塞非阻塞指的是加载文件。在很多的api中，我们可以看到一个接口经常有2种形态：

readFile

readFileSync

或者unity里面加载资源的接口：

Resource.Load

Resource.LoadSync

我们用js写文件的时候，很容易建立一个认知。当我们想对文件进行有序的操作的时候，我们用阻塞的方式。如果我们不需要任何的时序或者只使用到很少的时序，那么我们就可以用非阻塞的形式。

在游戏里面，经常会被建议不要用阻塞的方式。因为阻塞行为一旦发生，它是可能卡住整个界面的，使得游戏无法做出任何的响应。那我们是不是就不用阻塞了？我们先来看一看阻塞的优点。阻塞相对非阻塞而言，它的速度是比较快的。它可以在一些场景中进行应用：

在游戏的启动加载过程中，为了缩短玩家的等待时间。我们大量使用阻塞，只在单个文件加载后更新一下界面，接着就继续阻塞加载了。

在大部分游戏引擎里面，加载界面资源我们常用的是非阻塞。但是当我们加载界面资源后，如果实例化这个对象一定是耗时的或者说卡的。那么这个时候我们一般会先把界面的外框加载出来，再加载里面的内容，这个时候我们就可以换用阻塞了。当存在某些逻辑是不得不卡住的或者说耗时严重的，而且也无法通过线程或者其他方式优化的时候，我们先阻塞加载静态的背景，再进行一系列耗时的操作，包含阻塞加载资源，对象的实例化等。

这2个案例告诉了我们它的应用场景，在效率至上的地方，以及在耗时已经超出的地方。因为如果一定要卡（耗时），那么就在一瞬间铺个静态背景卡个够。这边还可以举个例子就是场景切换后，需要加载大量的玩家对象。这些对象在加载的时候因为数量，也因为其他不确定的因素，可能是会导致CPU的高峰。那么我们就在切换场景的加载页面出现之前，先同步把它们都加载完。这样，进入场景后就不会再遇到它们导致的卡顿。

对于非阻塞，通常伴随着事件完成的回调处理。比如说加载一个文件，我们会在加载完成后回调到我们注册的函数。对于代码而言，阻塞的代码写起来是比较舒服的，不用各种回调嵌套，因为回调一旦多了，会引发“回调地域”的灾难，这在js里面有专门的介绍。我们可以通过协程来解决，模拟一个同步的写法。

14. 同步与异步

同步与异步经常被其他人混淆，以为是和阻塞非阻塞一样的概念。但是实际上同步异步的概念要更加广一下。

我们来看一下客户端发起一个资源（不一定在本地或者可能有更新）的整个过程。

客户端调用底层接口加载资源，底层接口判断这个资源不存在或者有更新，向服务器发起下载资源的请求，请求完成后回调客户端接口，客户端接口加载资源。

客户端接口加载资源的这个过程我们可以使用阻塞或者非阻塞。但是纵观这整个过程，它是异步的。因为它需要跟网络交互，这个时间是不确定的，不是马上完成的。

如果客户端的请求是：

客户端调用底层接口加载资源，底层接口判断这个资源存在，回调客户端接口，客户端接口加载资源。

这样整个过程是一气呵成的，都在同一句代码里面发生。从资源加载到回调，我们说这样的一个行为是同步的。

最后我们再来看客户端加载资源的这个过程，因为它可能发生同步也可能发生异步的情况，所以总体上我们称它是异步的过程（只要有一环是异步的，那么它就是异步的）。

同步的操作是经常存在在我们的代码中，我们写的大量代码都是同步的，所以我们能很好的维护它的时序。而异步的代码经常伴随着回调，协程等技术。那看起来我们只需要了解它们的概念即可，然而我们要更加关注的东西是，我们可以通过机制的设计，来保证一个异步的行为在某些时候可以同步的调用。这是非常重要的，当我们在考虑设计的时候考虑到这一点，意味着我们可以通过转同步的方式来降低代码的复杂度。因为异步的代码无论是回调还是协程还是其他，都多多少少会破坏代码的可读性。在一些多个异步请求发生的过程中，处理起来需要加多个变量去记录状态做多重判断。这时候，如果我们可以将其中的某些异步操作在机制上转为同步，那么我们就可以避免代码上面的灾难。

我们来举个例子，看看怎么把异步转同步来降低代码的复杂度。SDK的提供产生有一个广告接口，它是有调用次数限制的。我们的客户端界面要求在没有广告的时候要隐藏界面的广告按钮。从前面的限制条件我们可以得出，我们要判断是否展现广告按钮，它是一个异步请求网络的流程。所以我们的接口原来是：

```
function canLoadAd(){

}
```

变成了

```
function canLoadAd(fCanLoad){

    //调用SDK的接口，在它的回调中调用fCanLoad

}
```

我们原来希望canLoadAd是一个同步操作，现在不得已变成了一个异步操作，而且它还很奇怪传入了一个回调用于在判断完成后做操作，比如隐藏按钮等。首先这个接口违背了我们前面的接口设计原则，canLoadAd就应该是个简单接口，返回true或者false，任何多余的东西都不该传入。那么这时候我们就需要把它变成一个同步接口了。

我们在项目启动的时候手动调用SDK的接口获取到广告剩余次数，那么我们的接口就可以变成：

```
function canLoadAd(){

    return 广告剩余次数

}
```

这就完成了一个异步转同步的过程。当然，我们这边转同步的方式是通过优化逻辑来减少代码复杂度。

事实上，我们在语言层面有个概念，叫做协程。它可以把一些异步的操作通过同步的写法写出来。注意这边和我们这节说的概念是不同的，我们这节的同步是真实的在逻辑上是同步的过程。而协议的同步化写法本质上还是异步的。我们来看几个例子来感受一下协程的同步化：

unity中的协程：

```
private IEnumerator Test()

{

    WWW www = new WWW("www.baidu.com");

    yield return www;

    yield return new WaitForSeconds(8.0f);

}
```

本应该是WWW的回调之后进行 WaitForSeconds，现在挨着写了。

typescript:

```
    private async loginAndLoad(loadingView){
        await platform.login();
        await RES.loadGroup("preload", 0, loadingView);
        MyApp.instance.createScene(GameScene);
    }
```

登录与加载资源都是异步的，也有同步的写法写了。

15. 串行与并行

在很多系统中，我们都会遇到串行和并行的情况。串行指的是几个事情有序依次的发生，并行指的是几个事情同时发生，不存在先后依赖。先举几个例子来认识下这些情况：

在关卡系统中，策划会配置刷出3只怪物，当怪物数量为0的时候，又刷出3只怪物，这是一个串行的过程。但是策划也可能想要当其中某只怪物死亡的时候，刷出5只怪。这个就是一个并行的过程，它们要监听并行的条件，执行并行的逻辑，生成并行的过程。

在引导系统的前期，往往是强引导的模式。玩家只能一个一个界面，一个一个按钮的点击。这种引导的时序就是串行的，当前面的引导未结束的时候就不能跳到后面的引导步骤。到了引导后期，玩家已经基本知道游戏怎么玩了。但是还是有新的一些情况需要对玩家进行引导，比如说金币低于多少数量的时候，要引导玩家去获取金币。或者说是与其他玩家交战产生了伤兵的时候。这样的一个个未知的条件就构成了引导里面的并行部分。

在怪物的ai行为中，怪物从1个点走向另外一个点，并在那个新的点说话，说话结束后返回原来的地方，这是串行的过程。但是在走向那个点的过程中，可能会有玩家对它进行攻击，可能有其他怪物跟它交流，这些情况都是并行的，会引发这个行为的并行流程。

首先我们看一下串行和并行之间可以是什么关系：

并排的关系

并行是多个串行的一种组合

我们采用关系2来界定它们，并分析看看如何实现这2种模式。并行是多个串行的一种组合，那么串行就是更加基本的一个单元，我们先来实现这种基本单元。

我们拿关卡系统来举例。在关卡系统中，从串行的事件发生到响应的结果我们称为一个过程。这样的一个过程由2部分组成，分别为条件和行为。条件指的是触发后面行为需要满足的因素，行为指的是条件满足后触发的行动。条件的触发需要事件，所以我们要把条件具体需要监听的东西封装成一个又一个不同的事件。比如说我们有个条件是监听怪物的数量，那么我们就需要抽象一个怪物数量变更的事件。如果我们有个条件是指定怪物死亡，那么我们就需要抽象一个怪物死亡的事件。在事件的基础上，条件就是对应着事件的监听。但是条件又不仅仅是这样，条件还具备自己的满足或者说达成关系。

假设我们有3个条件A,B,C。那么我们一定会产生疑问：到底是ABC都满足的时候触发后面的行为呢，还是ABC只需要满足一条的时候产生后面的行为。先满足A，再满足B，最后满足C的方式是否能触发后面的行为呢？

对于关卡系统而言，我们最终是采用了ABC都满足的时候触发后面的行为这种方式。选用这种方式的原因很简单，经过实践，普遍策划的逻辑能力可以在这种行为方式下不犯

错。而其他的条件结合模式，虽然有些在程序上的实现会很酷，但是很可能导致策划配置的错误（曾经使用ABC先后满足模型设计的关卡系统就导致了新手策划比较高的理解门槛）。

条件满足后，会触发行为。我们的行为很简单，就是串行的。在单个过程中，我们希望所有的条件，行为都保持简单的模型，不出现并行这样的逻辑，即使可以（比如同时执行多个行为）。我们可以把并行这块交给过程的并行来做，而不是放在过程内来实现。行为简单的从上往下就结束了吗？我们需要关注到显示生活中一个串行的事件，它其实不是一下子就全部发生的。比如你去买菜，你需要花5分钟踩单车过去，再回来。买菜到踩单车到回来不是一个瞬时发生的行为。所以，我们在串行的行为中，经常会加入一个延迟的指令，来模拟游戏中真实的时间流逝。

说完了过程本身，接下来来看我们原来讨论的并行。并行是同一时间多个过程在并发执行。意味着关卡中可能出现了第一个过程的怪物，也可能刷出了第二个过程的怪物。但是这边我们说的不行并不是真的并行，只是并发，所以多个过程之间还是需要有个顺序来规范谁先谁后的。这个问题我们放在编辑器中解决，在编辑器中从上往下就是它们的执行顺序。从上往下是人阅读的顺序，所以我们也采用了这个顺序。

对于引导，也是可以设计成和关卡一样。它在条件和指令上面支持的种类会少一些，但是总体框架是一样的。

对于ai，会有一些不同。在设计系统的时候，尤其是做系统策略的时候，我们需要考虑到面对的人。从某些角度上面，我们觉得配置ai的策划的逻辑理解能力会高于配置关卡的策划的理解能力。那么在ai的条件和行为部分都会复杂于关卡。

在ai中，它的条件模式是可选的，可以选择条件是全部满足触发，也可以是满足其一就触发，也可以对条件进行求反等操作。它的行为也支持不同的模式，可以是串行执行，也可以是随机行为执行。

16. 存数据引发的思考

项目组的小A接到这样一个需求：每日单笔充值达到配置金额，即可领取相应的奖励。配置的金额有好几档，比如达到10元可以领取A奖励，20元可以领取B奖励，30元可以领取C奖励。单笔达到30可以领取前面3档的奖励。隔天会自动发送未领取的奖励。

小A看了需求立马在模块对应的proto文件中定义了一个当日充值最高金额，洋洋得意准备开撸代码。等一下，老A拦住了他。老A说到：这个地方我们存最高档次也是可以的，我们来对这块存储的内容进行下推论。

存储最高金额有哪些优点？

在这个系统里面，最高金额是个相对的不变量，即使配置变更，极端点全部删除了，它依然不需要特殊处理，正常通过算法运作。

存储最高金额有哪些缺点？

如果第二档的金额从20变为了21，那么原先未领取奖励的玩家可能就无法领取了，也白白充了一次20.

存储最高档次有哪些优点？

如果策划对某个档次，比如第二档的金额进行微调，丝毫不会影响已购买玩家的体验，玩家该得的还是会得到。

存储最高档次有哪些缺点？

如果配置层面删除了对应的档次，我们可能需要写代码去调整最高档次。另外如果策划调整的不是简单的金额数值，而是需要插入1档，比如1,2直接插入1档15元的奖励，那么我们的做法可能会和预期不符（预期是充了20元可以领3档）。最不符合的是调整了档次的顺序，那么我们的存储也就乱了。

从上面的第一次PK中，存储金额和存储档次各有有缺点。我们需要继续讨论，尝试找到其中的微小的压倒性的理由支持我们去选择正确的数据存储。

思考2个问题：

1. 策划对金额的调整，是微调多还是调整档次多？

2. 2种做法玩家的反应是什么？

上面的第一个问题不一定可以立刻定下来，需要根据项目策划的总体水平进行猜测性评估，暂时我们只能说这是个平手。第二个问题中的调整档次的顺序在这边的可能性比较

小，因为档次总是从低往高去配置，这种情况也先给金额微微加点分。另一方面，当第二档金额从20变为21的时候，玩家是会抗议的，因为他可以可见性的发现他失去了原本可以领取的奖励。当多了一档金额时，玩家依然可以拿到原来的奖励，只是觉得少拿了。从另外的一个买东西的角度看，一种玩家是应有的东西少了，第二种是买的东西降价了。第一种的体验更加不好，所以我们这边选择了存档次而不是存金额。小C在旁边听着，那我也用上档次优化优化吧，小C说到。等一下，老A拦住了他。小C的需求和小A不同，他的需求是这样的：

每日累计充值达到配置金额，即可领取相应的奖励。配置同样有好几档。比如达到10元可以领取A奖励，20元可以领取B奖励，30元可以领取C奖励。累计充值达到30元可以领取前面3档的奖励。隔天会自动发送未领取的奖励。

这个需求，每日累计的充值金额是一定要存的，那么充值金额达到的最大档次是否要存储呢？

小C做出如下推论：

存储最高档次的优点？

如果第2档从原来的20元调到25元，那么存入档次可以使得玩家不会有机会获得两份2档的奖励。也可以使得原来累计充值达到20元的玩家可以有可能获得第二档的奖励。

存储最高档次的缺点？

需要额外维护一个最大档次的逻辑，容易引入额外bug。按照1的逻辑，如果把第二档删除了，那么只充值了20元的玩家可以直接领到30元的奖励，这不符合原来的设计。同样插入1档15元的，也会使得玩家领不到对应20元的档次，体验很奇怪。

这个设计的缺点是否能被克服呢？老A启发性的提问到。小C埋头沉思了一会，他想到一个方案，就是策划对档次不进行真正的删除，而只在对应的配置标记该档不启用，维持最高档次不变。而对于插入，则通过累计金额重新校正最大档次。

老A想起了他以前处理过的开放系统，当已开放的系统因为策划的改动而变得不满足开放条件时，提供了2种策略供策划配置，允许策划自由选择是否维持开放或者是重新判断。

老A的决定：

小C是勤于思考的，即使最终的策略有误，他也能想到方法去补救。所以可以由着小C自己决定了。

当然，不同的情况还是需要不同的推论了。比如小D遇到了一个需求：

每天达成不同的目标可以获得不同的奖励，策划可配置多个目标，且可配置目标的显示顺序。

考虑我们如何来存储目标的信息情况（可领取，未领取，已领取）。我们继续上面的推论，这边我们考虑可以存2种东西，1依然是把每个目标分成不同的档数，存档数对应的信息情况。2是存目标对应的Id的信息情况。我们做个推论：

存储档次的优点？

即使目标对应的Id改变，也能获得正确的信息状态

存储档次的缺点？

如果该目标ID已经不用了，比如是这个ID对应的玩法已经不存在了，那么目标的信息情况就错乱了。本不该获得的奖励就可能被获得了。

存储ID对应的信息的优点？

无论是新增或者删除ID都不会对奖励的信息有影响

存储ID对应的信息的缺点？

如果奖励是跟着对应的档次而定的，那么的这种情况我们将无法维持原有的奖励信息。

在这个需求里面，新增或者删除ID的可能性是比需要保证对应档次的奖励更大的。另外因为策划要求可以配置显示的顺序，那么证明了调换档次的顺序是个较高概率的操作。在这种操作下，存储档次是会有问题的。更加坚定了我们选择存储ID的决策。

17. 事件通知

设想有如下的场景，A窗口创建了B窗口，并且A窗口希望在B窗口被点击关闭的时候干些事情。那么我们有2种实现方式：

1. B窗口提供一个SetCloseCallback的接口，A窗口创建B窗口后，通过这个接口注册一个关闭的回调。当B窗口的关闭按钮被点击的时候，调用已经被设置好的回调函数。

2. B窗口继承了事件管理器，A窗口监听B窗口的关闭事件，B窗口在准备关闭的时候派发关闭事件。

下面对它们进行一下比较：

对于第一种实现，我们需要在B窗口中加一个设置回调函数，一个回调函数变量，并且在关闭接口中调用这个变量，A需要调用设置函数。对于第二种实现，需要在B的关闭函数中加上派发事件的逻辑，并且需要额外在一个表中定义这个事件。那么我们可以看到它们的代码量差不多。

对于第一种实现，B窗口提供的接口可能是SetCloseCallback，C窗口提供的可能是SetCallback，没办法完全的统一。而第二种实现下面，这个事件也可能不统一。但是这边第二种实现有个微小的优势，就是事件名往往短于函数名，所以更加容易实现统一。

B窗口因为需要发送和派发事件，那么需要继承或者组合指定的类，A可能也需要，这是一个负担。

第二种实现需要额外一个文件定义事件，方便同样。

综合上面的比较来看，第一种实现似乎会快一些，但是它的统一性比较差。所以，这边建议的方式是在框架层，可以用第二种来实现，可以做到整体的统一。而在逻辑层，可以使用实现1来加快编码效率。当然，这里说的都不是绝对的，我们依然可以在逻辑层使用方式2来保证统一性。

当我们对比一些方案产生了差不多的感觉的时候，我们还可以将项目放大来比较它们的优势与劣势。我们讨论下面的一种情况，A类使用了B类，B类使用了C类。我们做个真实的假定，A类是我们的逻辑类，B类是场景类，C类是怪物类。策划可能想，在逻辑类这个玩法中，所有的怪物的攻击数值（通常称为攻击力）都会根据一个动态的逻辑因子而改变。这个时候我们需要的是逻辑类能够监听到怪物类的创建，并且在怪物类创建的时候动态修改它的攻击力。如果是用设置回调的方式来实现，场景类需要提供一个怪物创建的回调接口，怪物类也需要提供一个自身创建的接口。我们可以很明显的看到，这样的方式是种重复。使用事件派发的形式实现的话，B只需要监听并继续分发该实现。这个监听并自动分发的行为在事件系统中是可以做的非常干净的，只需要配置即可实现。

接下来我们来看一种实现，服务器端的事件实现。先介绍下服务器端的一些基础框架：

有个玩家类Player，玩家类中存在多个玩家模块类PlayerModule，这些模块对应着玩家基础数据，vip，任务，邮件等。有了之前的基础，小C决定来实现这个事件系统。他遇到的需求是这样的：

等级变更的时候解锁新的任务。

原来的系统中存在的两个模块，等级模块与任务模块。小C的编写的代码如下：

```
class EventManager{
    public AddEvent(nEventId, fHandle)
    public RmEvent(nEventId , fHandle)
    public DispatchEvent(nEventId, …)
}
class LevelModule extends EventManager {
    private nLevel:number;
    function LevelChange(){
        this.DispatchEvent(nEventLevelChange, this.nLevel);
    }
}
class TaskModule extend EventManager{
    public OnCreate(){
        this.AddEvent(nEventLevelChange, this.onLevelChange);
    }
    private onLevelChange(){
}
}
```

代码实现了，功能也能正常运转。可是却有个很大的问题，那就是每个玩家的TaskModule都注册了一次，造成了无用的内存浪费。事实上，对于模块而言，我们只需要静态的注册一次。当事件发生的时候，只是事件的参数变了。我们将这块修改一下：

```
class EventManager{

    public AddEvent(nEventId, fHandle)

    public RmEvent(nEventId, fHandle)

    public DispatchEvent(nEventId, …)

    public static AddStaticEvent(nEventId, fHandle)

    public static RmStaticEvent(nEventId, fHandle)

    public static DispatchStaticEvent(nEventId, oObject)

}

class LevelModule {

    private nLevel:number;

    function LevelChange(){

        EventManager. DispatchStaticEvent (nEventLevelChange, this.nLevel);

    }

}

class TaskModule {

    public onLevelChange(){

    }

    }

EventManage.RmStaticEvent(nEventLevelChange, onLevelChangeWrapper);

EventManage.AddStaticEvent(nEventLevelChange,onLevelChangeWrapper;

function onLevelChangeWrapper(oObject){
```

```
    let oTaskModule = oObject:getModule(nTaskModuleId);

    oTaskModule:onLevelChange();

}
```

解释下上面的改动：

服务器的事件管理器新增了静态的接口，来解决内存的问题。当然，在服务器的一些动态的应用上，比如动态创建怪物，监听怪物的事件这样的，我们依然可以使用继承的方式。

上面的模块在注册前先反注册了接口，是为了防止热更新导致重复的注册。

事件系统本身应该具备静态与非静态接口，这对于服务器，客户端都是如此。在客户端，静态接口可以方便解耦一些完全没有关联的类。非静态接口可以解决使用，组合这种关系的类的通信，结合起来可以事半功倍。

18. 接口设计

我们在上面谈过了接口的参数设计，以及如何让接口可以自由的拓展。这边我们讨论另外一个东西，就是一些稍微复杂的接口应该设计成什么样。我们考虑定时器管理器的接口，先列出它的接口可能设计的样子，这边把this(oObject)也当做一个必要的参数。

a的设计：

AddTimeHandle(nTime, fHandle, oObject) 返回定时器oHandle

DelTimeHandle(oHandle) 参数是前面返回的定时器oHandle

b的设计：

AddTimeHandle(nTime, fHandle, oObject) 返回空

DelTimeHandle(fHandle, oObject)

a设计无论在AddTimeHandle需要几个参数，它最终都返回一个handle对象（句柄），删除的时候也是通过这个对象进行删除。b设计以fHandle,oObject为key，删除的时候也是传入这2个东西进行比较删除。接下来比较一下它们的优缺点：

上面的a方案返回的是一个handle对象，这个对象是可以存储很多信息的，也可以有很多行为。比如说取消这个定时回调，暂停这个定时回调。也可以设置它的执行次数等，它把一些可能需要在接口层定义的功能放到了这个对象内。

上面的b方案的优点是更加便捷，调用AddTimeHandle的类或者函数，不需要再去用一个对象保存它的返回值。

a方案还有一个额外的优势，它可以对一个fHandle注册多次。而在b方案中，fHandle与oObject构成了唯一的key。当这两个值一样的时候，就不能注册多次了，会触发重复注册的断言。

如果是在项目的规模比较大的情况下，我会推荐使用a方案来实现客户端。我们描述一下它的正确的用法。我们需要封装一个对象，对AddTimeHandle的返回handle做管理，放到一个数组里面。在一个生命周期结束的时候，比如说一个窗口，它的关闭就是生命周期的结束。这时候，我们需要在框架中去保证这个对象的销毁接口能正常进行。这样，我们就能保证即使逻辑层忘了释放对应的定时器，窗口也会帮它处理干净。这点，在b方案中是肯定无法实现的，所以我们在大规模项目中建议采用a方案，虽然需要多封装一层使用，但是它的价值是无穷的。

在上面的一节，事实上我们采用的是b方案来设计接口的，它的直观感受是很舒服不需要去管返回的handle。所以对于这一块，依然是要根据使用的规模来决定如何设计接口，没有固定套路。

最后有一个关键的地方，如果我们确定我们的接口设计是使用b方案的设计，那么我们应该确保底层实现中是通过fHandle，oObject来生成唯一key来进行相关操作的，而不是通过遍历匹配相等去操作，那样的话效率就得不到保障了。

19. 统一与非统一

统一指的是使用同一种实现方式处理看起来差不多的东西或者对象。非统一则是相反概念。我们举个例子说明：

一个玩家可以带上一只宠物和一只精灵，精灵和宠物都是跟随着玩家的。精灵是可以参与到战斗中的，要求我方看到的精灵和对方看到的精灵表现是一致的。通常情况下这个精灵会在服务器端实现主要逻辑，比如移动，释放技能等，并广播给看到它的各个玩家。宠物也是可以参与战斗的，但是它有一些特殊性。它无法被选中攻击，只会隔一段时间释放一个给玩家加血的技能。因为无法被选中攻击，所以不要求它在所有人的显示上是一致的。这时候就会出现2种做法：

a）考虑到宠物的位置实际上是不太重要的，也就是宠物的移动是不重要的。那么可以把宠物的移动放到客户端实现。这种情况可以减少大量的服务器广播导致的流量开销。移动的广播是非常消耗流量的，当玩家发生了一个移动时，需要把他的移动行为广播给所有看到他的玩家。因为次数的频繁，这种流量消耗的非常大。

b）把宠物的处理和精灵的处理一致化，统一使用一套东西实现，特化出宠物的逻辑。

这个问题不是效率与空间之间的比拼了，我们常说用时间换空间，用空间换时间。这里的问题是性能与统一之间的比拼。在这个问题上，我们可以这样分层，统一性是在时间与空间更上一层的东西，所以我们要先满足统一性， 在统一性的环节下做优化。比如上面这个例子，如果想减少广播量，那么我们可以在服务器控制不发送宠物的移动，依然由客户端自己去对宠物进行移动。

统一性带来的好处是可以良好的应对的统一化的需求。比如需求要求所有跟随都满足特定的逻辑，那么我们在服务器实现一套就行了，不需要在客户端再实现一套。虽然我们可以实现相同的算法，但是有时候数据并没有完全同步，这就会造成处理上的不同。

在统一与非统一上还存在着一些额外的思考，我们举个在api上面的例子。js里面有个对象是Date对象，它可以用来获取时间以及转换时间。这个对象在浏览器或者是nodejs的环境中都是存在这个对象的。正常情况下我们可以在客户端中使用，也可以在服务器（跑在nodejs环境里面）中使用。微信的云开发就是一个nodejs的环境，但是我们在阅读它的文档的时候，我们会发现它提供了一个serverDate这样的对象。这个对象和Date对象命名很相似，我们会认为它可能是个替代品。继续阅读我们可以知道这个特殊的对象实际上在存储的时候进行了效率上的优化。那么问题来了，我们应该使用统一的Date对象还是使用效率更好的serverDate对象呢？

在效率没有达到一定程度的影响的时候，我们推荐的是优先考虑统一性。统一意味着更高程度的复用，这是我们积累与抽象的基础。

20. 耦合与非耦合

耦合是指程序模块之间存在联系的紧密程度。在我们之前的分层小节中，我们强调的是尽可能的减少依赖，解除耦合。如何A事情的发生不需要B事情作为先决条件，那么A这件事情就相对而言更加独立。更加独立的目的是为了更容易划分，且并行执行。对于一件事情而言，A事情和B事情同时安排给两个人独立完成是比较高效的。而如果A事情的开始一定要等到B事情完成，那么这种效率就是比较低的。我们希望的减少耦合的本质目的是为了效率。

明白了解除耦合的目的，我们在很多地方就会尽可能让事情变的独立。然而，对于我们的现实生活，我们可以看到互联网的出现大幅度的提高了社会的生产力。信息的流通促进我们更好的发展，这种流通性或者说信息的透明折射到我们的代码之中，也给了非耦合这个思想一个大大的挑战。我们来看一个基于透明而设计的框架。

在众多的代码中，会出现一种框架写法是这样的：

```
class BaseClass{

    public getSingleTimeManager(){}

    public getMainView(){}

    public getBattleView(){}

    //=========================

}

class MainView extends BaseClass{

}

class BattleView extends BaseClass{

}
```

上面的框架是这样的，有个基础类，里面包含了你会用到的一切。其他的所有类都继承自这个基础类。所以，在这样的一个框架下，无论你处于哪个类中，你都能很好的访问到所有类的情况。

这种框架在小游戏中会一定程度的出现，因为小游戏的窗口数量，代码规模都不会太大。在小游戏的项目中，这个基类不会因为太多的逻辑而变的过大。但是到了大型的项目中，这种框架会导致一些问题：

1. 基类包含了所有的类的使用代码，当我们的类的数目增大的时候，基类的代码会变得非常的臃肿，以致于后期将无法维护。

2. 因为任何类以及对象之间都可以直接访问，那么当我们要删除一个类的时候，所有引用它的代码都需要删除，这是个繁琐的操作。

从代码的角度我们还是比较容易的知道，这种集中式的框架存在着一定的问题。但是我们反着又把它映射到现实生活，我们发现互联网这种将信息构建到一个地方供大家取用的能力并没有导致它无法维护。再细细整理起来，我们会发现互联网初期当数据变多的时候，它也是非常臃肿的。直到后面对它进行了合理的分类，分层后，它的复杂度降低了。它的分层有很多例子，比如说将原始数据划分一个层，将展示划分一个层，将处理数据的划分一个层。所以，集中式的复杂度的解决问题还是回到了分类，分层。所以，我们不如一开始就设计一个低耦合的系统。

面向对象有一些原则：高内聚，低耦合。多聚合，少继承。哪些是耦合呢？

1. B是A的属性，或者A引用了B的实例。

2. A调用了B的方法。

3. A直接或者间接的成为了B的子类。

4. 元素A是接口B的实现。

耦合不一定都是不好的，在后面的框架代码我们可以看到，一定的耦合是允许的。因为逻辑代码是易变的，所以逻辑代码应该尽量减少耦合。

第三章

框架设计初步

1. 基类的设计

基类Base类是我们构建框架的基础类，是所有后续整个框架中所有类都应该具备的能力。不一定要继承，但是一定要有这样的接口。

Base是万物的根本，它提供着一个和万物一样的基础属性，也就是生与死。它需要一个变量destroy来实现标记销毁，销毁即为死，非销毁即为生。

Base提供了2个基础的接口

```
abstract Base{

    isDestroyed();

    isValid();

}
```

让我们再想想，还有什么东西是基础性的。对比人来看，每个人除了生与死的状态，还具备唯一性。那么为了表示唯一性，我们需要再加一个uniqueId这样的变量，并且提供GetUniqueId这样的接口给外界，Base调整后变成这样。

```
Base{

    isDestroyed ();

    IsValid();

    getUniqueId();

}
```

我们不提供SetUniqueId，是因为这个uniqueId是与生俱来的，由系统机制实现的。同样的，在这个唯一Id的基础上，我们可以提供比较是否是同一对象等接口。

当然，如果只是上面这样2个变量，我们可以把2个变量合并成一个变量，从而节约内存。注意我们的基类的设计一定是简洁且符合效率的。

知道了一个对象的生死，我们就可以在很多异步回来的时候进行判断，避免逻辑上的错误。举个例子，有个窗口在打开的时候发起了一个网络的异步请求，当请求完成时要修改窗口里面的一张图片。但是这个窗口在消息回来之前可能已经销毁了，这个时候就不能去操作里面的图片了，因为对象已经不存在了。所以，我们需要知道对象的生死情况，才能更好的处理各种异步的情况。

2. 框架代码结构

很少有地方会讨论框架的代码结构，通常是以我们要建立的框架类型，比如ui框架等，直接去讨论类与类直接的关系，类的成员等。先看下我们推荐的框架代码结构，以ui管理器为例。

文件的最前面一部分是注释

```
/*ui管理器，管理若干个窗口对象*/
```

注释里面需要写清楚作用，以及设计思想，以及后续的拓展计划。

第二部分是类名以及它的成员

```
class UIManager extends Base {

    private tUIViews = [];

}

class UIView extends Base {

    private nSort = 1;

}
```

第三部分是接口，这是这里面最重要的部分。

接口分为3个部分：

```
//子类关注的接口

//框架实现接口

//框架子类关注的接口

//私有接口

//对外接口
```

子类关注的接口并不一定是外界无法访问的，只要子类可能对它存在拓展需求，我们就把它归为子类关注的接口。这边我们首先要明白，在一个框架里面，存在管理者和被管理者，它们都要考虑被其他对象继承。对于UIManager而言，它的debugName应该是子类关

注的，因为子类可能会拓展它，定义不同的调试名称。对于UIView而言，它的onCreate接口是每个子类需要关注的。这里的子类关注也可以对应成一些语言里面的虚函数。

框架实现接口指的是为了实现这样的一个框架而开发的接口。比如说loadSkin 这样的接口，这个接口只有UIManager在创建的时候会使用，这种接口我们就把它放到框架实现接口里面，这种接口虽然是公有的，但是外界不应该直接去调用。

框架子类关注的接口，与子类关注的接口不同的是，这样的接口的目的是用来拓展框架的，而子类关注的接口是服务于逻辑的，由逻辑覆盖实现的。

子类关注的接口和框架子类关注的接口有一点非常大的区别在于：

子类关注的接口基类中一定是个空的实现，意味着子类不需要考虑去调用父类同名接口。而框架子类关注的接口一定要在实现中调用父类的同名函数。对应逻辑层，是减轻逻辑层的使用负担。对应框架的拓展者，要求它保证整个调用链的完整。当你在设计框架的时候，如果需要逻辑类去调用同名接口，比如：

```
function onCreate(){

    Base.onCreate();

    //干其他的事情

}
```

这种就是不好的设计，因为使用者经常会忘记调用父类的接口，导致很多问题的发生。所以，这样要区分这2种接口。

当某个程序需要实现一个buff的显示界面时，他要做的事情就是拉到uibase的最上面，看一下那些子类关注的接口就行。当当前的ui框架无法实现项目的需求的时候，他需要拉到框架子类关注的接口部分去尝试拓展。

私有接口顾名思义，就是只有当前对象才能访问的接口。

对外接口也是很明显，列出的就是所有的对外接口。再次注意，上面的接口有些也是开放可以调用的，但是可能因为语言本身的限制，无法做到只对部分类的访问设置访问权限，所以这些接口我们不应该去调用（放在整个项目的规范内）。

上面的接口设计需要确保理解的情况下才能进入下一节。接口设计的理解保证了即使无法合理的开发一个框架，也能合理的修改框架。

3. 框架设计

本节我们要开发一个UI框架，底层以白鹭引擎为例。

框架设计的第一步并不是直接撸代码，而是先想清楚设计思想，抽象。

一个一个的UI窗口是独立的吗？不是的，因为它有层级关系，A窗口在B窗口下面，新创建的C窗口在最上面。因为它们直接有关系，所以它们需要被管理。另外一个原因是：统一的管理有利于机制的实现。举个例子，我们希望在大部分窗口打开的时候都有一个从小变大的效果，那么我们可以通过窗口管理器提供机制去支持这个需求，并且只要改动少量的不需要的这个特性的窗口的代码就可以。这边我们说清楚了为什么要做管理，接下来我们讨论的是管理是不是一个普遍行为。

管理发生不仅仅发生在代码中，也发生在我们生活的方方面面。老板管理高层员工，高层管理中层，中层管理下层员工，这都是管理。同样的，在代码里面，也有很多管理。窗口管理器管理窗口，场景管理器管理场景，所以这种管理就是一种高度的抽象，可以抽象成一个通用的机制。我们现在谈的UI框架，实际上就是一种管理思维，它可以作为窗口管理器管理窗口，也可以作为场景管理器管理场景。我们先过一眼这个框架：

UIManager.ts

```
class UIManager extends Base{

    private tUIObjs:Object = {};

    private oCanvas:egret.DisplayoUIObjectContainer;

    public constructor(oCanvas：egret.DisplayoUIObjectContainer) {

        this.oCanvas = oCanvas;

    }

    /////////////// 子类关注接口 /////////////////

    //ui对象准备开始加载

    protected onUIObjBeginLoad(oUIObj:UIView){

    }

    //私有接口

    private destroyoUIObj(oUIObj:UIView){
```

```
            oUIObj.onPreDestroy();//新增的
            oUIObj.onDestroy();
            this.oCanvas.removeChild(oUIObj);
        }
        private loadUIObj(oUIObj:UIView){
            let sSkinName = oUIObj.skinXMLName();
            if(sSkinName === undefined){//不需要加载皮肤
                this.uiSkinReady(oUIObj);
            }
            else{
                oUIObj.addEventListener(Event.SkinReady,
    this.uiSkinReadyEvent, this);
                oUIObj.loadSkin();
            }
        }
        private uiSkinReadyEvent(event:Core.Event){
            this.uiSkinReady(event.data);
        }
        private uiSkinReady(oUIObj:UIView){
            this.oCanvas.addChild(oUIObj);
            this.onUIObjBeginLoad(oUIObj);
            oUIObj.setLoaded();
            oUIObj.registeEventMap();
```

```
        oUIObj.onPreLoaded();
        oUIObj.onLoaded();
        oUIObj.onEndLoaded();
    }
    /////////////// 对外接口 /////////////////
    //窗口创建接口
    public createUI(sClassname:string, ...args: oUIObject[]):UIView {
        let oUIObj = <UIView>this.tUIObjs[sClassname];
        if(oUIObj){
            return oUIObj;
        }
        let oClassType:any = egret.getDefinitionByName(sClassname);
        oUIObj = new oClassType(args);
        this.tUIObjs[sClassname] = oUIObj;
        oUIObj.setName(sClassname);
        oUIObj.onCreate();
        this.loadUIObj(oUIObj);
        return oUIObj;
    }
    public destroyUI(sClassname:string){
        let oUIObj = <UIView>this.tUIObjs[sClassname];
        if (!oUIObj) {
            console.warn("DestroyUI not exist sClassname:" +
```

```
    sClassname);
                return;
        }
        this.destroyoUIObj(oUIObj);
        delete this.tUIObjs[sClassname];
    }
        public destroyAllUI(){
            for(let uiName in this.tUIObjs){
                let oUIObj = <UIView>this.tUIObjs[uiName];
                this.destroyoUIObj(oUIObj);
            }
            this.tUIObjs = {}
        }
    }
```

UIView.ts

```
    class UIBase extends eui.Component {
        private bLoaded: boolean = false;
        private bDestroyed = false;
        public constructor() {
            super();
        }
        ////////////// 子类关注的接口 //////////////
        //窗口对象创建时
```

```
public onCreate(){}
//所有东西都加载完成
    public onLoaded() { }
    //销毁时
    public onDestroy() { }
    //事件映射表
    public eventMap() {
        return undefined;
    }
//皮肤名称
public skinXMLName():string{
    return undefined;
}
////////// 框架子类关注的接口 //////////
//准备加载
    public onPreLoaded() {
}
public onEndLoaded(){
}
////////// 框架使用的接口 //////////
public setLoaded(){
    this.bLoaded = true;
```

```
        }

        //皮肤加载接口

        public loadSkin(){

            this.addEventListener(eui.UIEvent.COMPLETE, this.onSkinComplete,th
is);

            this.skinName = "resource/eui_skins/view/" + this.skinXMLName() +".
exml";

        }

        private onSkinComplete(){

            this.removeEventListener(eui.UIEvent.COMPLETE,

    this.onSkinComplete, this);

            let oSkinEvent = egret.Event.create(Event, Event.SkinReady, false);

            oSkinEvent.data = this;

            this.dispatchEvent(oSkinEvent);

            egret.Event.release(oSkinEvent);

        }

        public registeEventMap(){

            let tEventMap = this.eventMap();

            if(tEventMap == undefined){

                return;

            }

            for (var i = 0; i < tEventMap.length; ++i) {

                let tEventInfo = tEventMap[i];

                let oComponent = <eui.Component>tEventInfo[0];
```

```
                let fFunc:Function = <Function>tEventInfo[1];
                let sEventType:string= <string>tEventInfo[2] ||egret.TouchEvent.TOUCH_TAP;
                oComponent.addEventListener(sEventType, fFunc, this);
            }
}
        public setDestroy(){
            this.bDestroyed = true;
        }
        //对外接口
        public isValid(){
            return !this.bDestroyed;
        }
        //私有接口
        private isLoaded(){
            return this.bLoaded;
        }
        private isDestroyed(){
            return this.bDestroyed;
        }
}
```

这个框架的基础部分的接口设计使用前面介绍过的设计方式。总的设计逻辑是这样的：

收集资源加载所必须的api，确定是同步加载还是异步加载。这边使用的loadskin是异步加载。

封装基础时机。基础时机在这边指的是一个被管理对象的创建和销毁以及异步加载完成。最本质的是创建以及销毁，及onCreate和onDestroy接口。异步加载完成时机是因为我们选用了异步加载而额外多出的时机（onLoaded）。在加载资源的各个阶段将这个”消息”合理的告诉子类，对于管理器而言，它告诉了子类准备开始加载子对象（onUIObjBeginLoad），而被管理者UIView告诉子类很多信息，比如onCreate，onDestroy，onLoaded。

加强被管理器的基础功能，抽象出配置化编程基础。

添加自动化平衡处理，防止逻辑错误。

这边解释下上面的点，onCreate和onDestroy在机制这层是一定保证对称出现的。即使继承类出现了逻辑异常，也需要保证它们都会各执行1次。配置化编程是这里面的设计的另外一个核心点，配置化编程指的是我们写代码的时候采用的就像配置一样的方式编码，任何冗余的代码都不需要书写。举个例子，如果你需要监听按钮A和按钮B的两个事件，你这么写：

```
this.oBtnA.addEventListener(egret.Event.TouchEvent, this.clickBtnA, this);

this.oBtnB.addEventListener(egret.Event.TouchEvent, this.clickBtnB, this);
```

这里面不同的是oBtnA，clickBtnA 与oBtnB，clickBtnB，其他都是相同的，所以我们用配置化编程改一下，变成了，我们的子类只需要实现：

```
public eventMap() {

    return [

    ["oBtnA", "clickBtnA" ],

    ["oBtnB", "clickBtnB" ],

    ];

}
```

ok,我们的重复代码就消除了，这就是配置化编程的核心处理。

自动化平衡处理指的是在一个被管理者的生命周期，它要保证在最后能把该释放的都释放了，防止逻辑层泄露。举个例子，一个ui里面使用了一个定时器，通常的写法是在onDestroy里面判断存在就去销毁它，但是很多程序可能会忘记释放。那么这时候我们需要封装定时器管理对象，在准备销毁前去调用该对象的清理函数，保证所有的定时器都能被正确销毁。

这个定时器对象在创建定时器后都会保存住创建后的句柄，如果逻辑层没销毁，那么最后机制会统一帮它清理。自动化平衡处理还能干一些事，比如说异步加载资源回来，逻辑层不需要再去判断窗口是否销毁，回调的接口一定是保证了窗口存在的情况下才调用的。这同样也需要封装一个异步资源管理器对象来实现。

下面我们来看4个东西：

基于上面的代码，如何封装游戏项目的场景管理器和窗口管理器。

实现一个自动化平衡处理。

实现一个窗口

感受机制的威力

封装场景管理器

```
class SceneManager extends UIManager {

    private oCurScene:SceneBase;

    public constructor(canvas:egret.DisplayObjectContainer) {

        super(canvas);

    }

    //子类框架实现接口

    protected onUIObjBeginLoad(oUIObj:UIView){

        if(this.oCurScene != undefined){

            this.destroyScene(this.oCurScene.getName());

        }

        this.oCurScene = <SceneBase>oUIObj;

    }

    //对外接口

    public createScene(sClassname:string, ...args: Object[]):SceneBase{
```

```
        return <SceneBase>this.createUI(sClassname, ...args);
    }
    public destroyScene(sClassname:string){
        return this.destroyUI(sClassname);
    }
    public createWindow(sClassname:string, ...args:Object[]):WindowBase{
        Core.assert(this.oCurScene != undefined);
        return <WindowBase>this.oCurScene.createWindow(sClassname, ...args);
    }
    public destroyWindow(sClassname:string){
        Core.assert(this.oCurScene != undefined);
        this.oCurScene.destroyWindow(sClassname);
    }
    public getWindow(sClassname:string):WindowBase{
        return this.oCurScene.getWindow(sClassname);
    }
    public getCurScene():SceneBase{
        return this.oCurScene;
    }
}
```

封装场景：

```
class SceneBase extends UIView {
    private oUIMgr: UIManager;
```

```
    //子类框架实现接口
    public onPreLoaded(){
        this.oUIMgr = new UIManager(this);
    }
    public createWindow(classname:string, ...args:Object[]): WindowBase{
        return <WindowBase>this.oUIMgr.createUI(classname, ...args);
    }
    public destroyWindow(classname:string){
        return this.oUIMgr.destroyUI(classname);
    }
    public getWindow(classname:string):WindowBase{
        return <WindowBase>this.oUIMgr.getUI(classname);
    }
}
```

封装窗口：

```
class WindowBase extends UIView {
    public constructor() {
        super();
    }
    //子类框架实现接口
        public onPreLoaded() {
            //窗口从小变大效果
        }
```

```
    }
```

上面封装的一些缘由：场景同时只能存在一个，窗口需要一些自定义的效果。因为有了这些特殊性，所以延伸了它们的子类。上面还有一个地方特别要注意，子类关注的东西如果被继承类实现了，那么这个接口在它子类这个级别就变成了框架子类关注，因为子类要在实现的时候调用父类同名接口了。

下面来看一个自动化平衡处理，处理下定时器。

先加强下框架的功能：

在UIManager的代码中加一个代码

```
    //私有接口
    private destroyoUIObj(oUIObj:UIView){
        oUIObj.onPreDestroy();//新增的
        oUIObj.onDestroy();
        this.oCanvas.removeChild(oUIObj);
    }
```

在UIView的框架子类关注的接口中加一个

```
    public onPreDestroy(){
    }
```

TimeHelp的实现：

```
class TimeHelp {
    private tHandlers: Array<TimerHandler> = new Array<TimerHandler>;
    public addTimeHandle(){
        let oTimerHandler = ; //调用引擎时间管理器接口返回oTimerHandler
        tHandlers.Add(oTimerHandler);
        return oTimerHandler;
```

```
    }
    public rmTimeHandle(oTimerHandler){
        if(oTimerHandler.isValid()){
            //调用引擎时间管理器接口移除oTimerHandler
        }
    }
    public clear(){
        let tHandlers = this.tHandlers;
        for(let i = 0; i < tHandlers.length; ++i){
            let oTimerHandler = tHandlers[i];
            if(oTimerHandler.isValid()){
                //调用引擎时间管理器接口移除oTimerHandler
            }
        }
        this.tHandlers.clear();
    }}
```

最后在windowbase里面集成：

```
class WindowBase extends UIView {
    private oTimeHelp:TimeHelp = new TimeHelp();
    public constructor() {
        super();
    }
    //子类框架实现接口
```

```
        public onPreLoaded() {
            //窗口从小变大效果
        }
        public addTimeHandle(...){
            //调用系统时间管理器接口返回oTimerHandler
            return oTimeHelp.addTimeHandle();
        }
        public rmTimeHandle(oTimerHandler){
            oTimeHelp.rmTimeHandle(oTimerHandler);
        }
        public onPreDestroy(){
            oTimeHelp.clear();
        }
    }
```

集成完毕，一切都很安静，这就是防止逻辑犯错的平衡。而UI逻辑层的都调用新封装的接口而不是直接调用底层接口，这样就避免的犯错的可能。

接下来我们看一个窗口的实现，非常简单：

```
    class SettingPanel extends UIView{
    private oBtnUIEffect:eui.Image;
    private oBtnBg:eui.Image;
        //事件映射表
        public eventMap() {
            return [
```

```
            ["oBtnUIEffect", "changeUIEffect"],
            ["oBtnBg", "changeBg"]
            ];
        }
    //皮肤名称
    public skinXMLName():string{
        return undefined;
    }
    public onLoaded() {
        //显示默认设置
    }
    private changeUIEffect(){
    //切换ui音效开关
    }
    private changeBg(){
        //切换背景音乐开关
    }
}
```

干干净净，舒舒服服，代码很清晰。

上面主要是讲我们对于我们的使用者，逻辑UI的编写者提供了很大的便利，以及防止他们的一些逻辑错误，但是它的威力远远不止如此。

我们假定我们现在开发了很多个系统，突然策划提出我们需要在按钮的点击响应上面播放点击音效。这时候有个做法是在每个事件响应的回调里面添加一句播放的代码。但是，自

从我们有了机制，有了对消息的拦截，我们可以在事件响应的地方做拓展。将UIView的registeEventMap拓展下：

```
public registeEventMap(){

        let tEventMap = this.eventMap();

        if(tEventMap == undefined){

            return;

        }

        for (var i = 0; i < tEventMap.length; ++i) {

            let tEventInfo = tEventMap[i];

            let oComponent = <eui.Component>tEventInfo[0];

            let fFunc:Function = <Function>tEventInfo[1];

            let sEventType:string= <string>tEventInfo[2] ||
egret.TouchEvent.TOUCH_TAP;

oComponent.addEventListener(sEventType, this.eventHook.bind(fFunc, this),this,
fFunc);

        }

    }

    private eventHook(fCallFunc){

        //播放声音

        fCallFunc();

    }
```

可以看到，一旦有了机制，我们就可以实现全局性的功能，尤其擅长处理大规模需要重复性代码的东西。这也是需要所有窗口都继承自UIView，且由UIManager创建的意义。

4. 框架拓展的思路

上面一节我们已经看到了平衡处理的艺术，这一节我们讨论的是新增一个系统集成到UI框架的过程。

第一步关注的基础能力，这个能力可以从引擎api中获取，比如api一开始就提供了声音播放的能力。也可以自己去实现这个能力，比如说在iphoneX的刘海在左右变换的时候提供事件通知。获取某个信息，值以及获取某个事件的变更，这些我们都称之为基础能力。基础能力是后续封装类的基础。

第二步是封装类，封装类的目的是解决接口到商用的问题。一个接口容易出现的是需要清理资源，需要显式赋值固定参数，重复调用类似代码等问题。这些问题都在封装类中进行解决，封装类基于配置化编程的思想，将所有的重复代码抽象出来。处理好资源的添加与释放，封装出适用于项目的接口和对象。

第三步是集成。将封装好的类集成到框架之中。这边不一定是UI框架，其他的框架也是一样的集成方式。集成保证了所有的请求都能正常的走到封装类中，便于后续做集中化的控制。

大致的过程如下面所示：

第一步基础能力

```
utilPlaySome(xxx)
```

第二步封装类

```
class PlaySomeHelp{

    playSome(){

        utilPlaySome()

    }

    clear(){

    }

}
```

第三步集成

```
class frame{
```

```
    private playSomeHelp:PlaySomeHelp = new PlaySomeHelp();
    playSome(xxx){
        playSomeHelp. playSome(xxx);
    }
    onDestroy(){
        playSomeHelp.clear();
    }
}
```

5. 路由

这边说的路由的指的是在代码执行的过程中，代码的控制权从一个类交给了另外一个类。在前面的UI框架中，也存在着路由，UIManager在创建一个窗口的时候，将消息或者说控制权路由到了窗口的onCreate接口，这就是一个路由。固定的路由就构成了机制的基础。

我们来看一个服务器路由的例子：

```
class Player{

    private tModules:Object = {};

    public addModule(sModuleName){

        let tType = //反射sModuleName获得类型

        let oModule = new tType();

        this.tModules.push(oModule);

        oModule.onCreate();

    }

    public registeModule(){

        return ["LevelModule", "TaskModule"];

    }

    public onCreate(){

        let tModule = this.registeModule();

        for(let i = 0; i < tModule.length; ++i){

            let sModuleName = tModule[i];

            this.addModule(sModuleName);

        }

    }
```

```
}

class PlayModule{

    public onCreate(){}

}

class LevelModule extends PlayModule{

}
```

这个例子上面的路由与UI框架的路由有点类似，也是一个自动化的路由，里面也使用了配置化编程。简单的路由就是这样，下面我们会谈谈更丰富的路由。在我们后台的接口中，经常需要用到路由功能。比如说 /admin 这样的访问需要路由到 admin文件夹对应的文件中，/api 这样的访问要路由到api文件夹中。我们可以看到，外面大部分说的路由其实是这种有规则的路由，并不是我们前面讲的简单路由。

那么在游戏中，同样也存在着这样的路由规则，但是远没有网页路由需要的那么复杂。

举个例子：玩家可能会参与到很多不同的玩法里面，同一时间只会进入一个玩法中，也就是这些玩法是互斥的。每个玩法都有对应的一个模块，每个模块和LevelModule一样的注册方式。这些玩法在玩家切换场景进入玩法的时候必须要被通知到。一种比较暴力的方式是我们在玩家切换场景进入玩法的时候把这个消息路由到所有的模块中，每个模块都判断当前的玩法是不是自己的玩法来决定是否进行处理当前消息。这种做法会导致很多无用的判断，虽然这个判断不一定会有很大消耗。还有一种更加错误的做法，就是在切换场景的时候主动去调用相应的接口，比如这样：

```
class Player{

    function onSceneChange(){

        this.getModule(tModuleId.eTask). onSceneChange();

        this.getModule(tModuleId.eFight). onSceneChange();

    //=====

        }

    }
```

这违反了前面的分层原则。

修改后的代码如下：

```
class Player{
    private tModules:Object = {};
    private tFuncMap = {};//新增
    public addModule(sModuleName){
        let tType = //反射sModuleName获得类型
        let oModule = new tType();
        this.tModules.push(oModule);
        oModule.onCreate();
        //新增
        this.tFuncMap[oModule.funcId()] = oModule;
    }
    public registeModule(){
        return ["FightModule"];
    }
    public onCreate(){
        let tModule = this.registeModule();
        for(let i = 0; i < tModule.length; ++i){
            let sModuleName = tModule[i];
            this.addModule(sModuleName);
        }
    }
    //新增的
```

```
    public onSceneChange(){
        let nCurFuncId = //获取funcId
        let oModule = this.tFuncMap[nCurFuncId];
        oModule.onSceneChange();
    }
}
class PlayModule{
    public funcId(){}
    public onCreate(){}
    public onSceneChange();
}
enum funcIds{
    eFight = 1
}
class FightModule extends PlayModule{
    public funcId(){
        return funcIds.eFight;
    }
    public onSceneChange(){
        //干自己需要的，一定是自己的玩法
    }
}
```

新增了一个数据结构tFuncMap 建立id到模块的映射，通过这个映射在适当的时机获取到对应的模块进行正确的路由。这就是游戏中通常用到的路由处理方式。

注意下这段代码：

```
public onSceneChange(){

    let nCurFuncId = //获取funcId

    let oModule = this.tFuncMap[nCurFuncId];

    oModule.onSceneChange();

}
```

这边也是可以派发事件的，但是我们这么写而不是使用事件是因为最好的消息通知方式是**当一个类有需要知道某个消息的时候，它只要实现一个接口，不需要做任何的注册反注册。**这种消息的路由方式是最提倡的，通常有底层框架去实现。需要接口的时候就实现，不需要的时候就直接删除。实现与删除就是注册与非注册的过程。

6. 其他框架的封装

前面主要讲的是UI框架的一些封装与实现。全局看我们的项目，只要我们做了足够多的抽象，我们就可以提炼出框架。我们在服务器端写代码的时候，也经常会看到同事们在自己的需求上使用管理器来统一管理数据。比如说组队需求通常会使用组队管理器来管理各个队伍。战场需求会使用战场管理器管理各个战场。直观的看这些需求里面的共性就是管理。管理这一部分和我们前面UI管理器做的这一部分是一样的，管理意味着对管理对象需要支持添加和删除，就像createWindow和destroyWindow这2个接口干的事一样。只有这些管理是不够的，接着我们还需要深入去提炼它们的共性。他们都会加载数据，保存数据，所以这也是一个共性。保存数据的时候会及时保存也会定时保存，这也是一个共性。反复看看大部分同事写的代码去提取框架的共性是一个好方法，这也是为什么写框架会比较难的一个原因，因为它需要有人其他代码去辅佐细化。

基于上面的共性，我们先写接口部分，把大体框架理出来：

```
class ManageredBase{
    //子类实现的接口
    public onLoad(){
    }
    public onSave(){
    }
}
class ManagerBase{
    //子类实现的接口
    public saveInterval(){
        //保存时长，0为立即保存
        return 30;
    }
    //对外接口
```

```
        public loadData <T extends ManageredBase>(c: new () => T) {

        }

        public saveData(){

        }

        public createData <T extends ManageredBase>(c: new () => T , ...args:O
bject[]){

        }

        public destroyData(){

        }

    }
```

接下来的一步就是关注基础能力了，因为加载数据的接口和保存数据的接口与具体使用的数据库有关，所以这边我们需要去深入了解这部分api的使用。应用api后的代码如下：

```
    class ManageredBase{

    //子类实现的接口

        public onLoad(...args:Object[]){

        }

        public onSave(){

        }

    }

    class ManagerBase{

        private tManageredData = new Array();

        //子类实现的接口

        public saveInterval(){
```

```
        //保存时长，0为立即保存
        return 30;
    }
    //对外接口
    public loadData<T extends ManageredBase>(c: new () => T){
        let tData = new Array();
        //使用api加载所有的数据到tData
        for(let i = 0; i < tData.length; ++i){
            let oData = this.createData(c, tData[i]);//通过数据构造对象
            this.tManageredData.push(oData);
        }
    }
    public saveData(){
        let tData = new Array();
        let tManageredData = this.tManageredData;
        for(let i = 0; i < tManageredData.length; ++i){
            let oData = tManageredData[i];
            tData.push(oData.onSave());
        }
        //使用api加载保存tData到数据库
    }
    public createData <T extends ManageredBase>(c: new () => T , ...args:O
bject[]){
```

```
        }

        public destroyData(){

        }

    }
```

接着是路由，将管理类收到的消息路由给对应的子类。上面ManageredBase 的onLoad的函数还没被调用，就是因为没有被路由。onSave在saveData中已经被路由了，下面看下onLoad的路由。

```
    class ManagerBase{

        public createData<T extends ManageredBase>(c:new () => T, ...args:Obje
ct[]): T{

            let oData = new c();

            oData.onLoad(...args);

            return oData;

            }

        }
```

还有个功能还没实现，就是关于保存间隔的。这部分的实现也是先了解底层定时器相关接口，在启动后注册定时器实现，这边就不具体展开了。

这样的封装使得数据的加载和保存有了统一的方式，管理器开发者不需要再考虑加载和保存相关的代码逻辑，只需要考虑保存的代码策略。继承自相应的基类后就可以拥有一个管理器的基础功能。

7. 框架思维应用案例

框架是可以解决很多问题的。在 iphoneX的机子上，出现了额外的一个刘海。如下图：

这个刘海使得我们的界面会被部分遮挡，有些功能无法正常点击。

对于这样的一个问题，我们要想的依然是如何在全局上解决这样的问题。首先要进行分析问题，刘海可能在左边也可能在右边。无论它位于哪一边，我们都需要对界面上的部分元素进行位置调整。通过上面的分析，我们可以做出规划：

1. 需要底层提供刘海朝向以及刘海变更的事件。

2. 当刘海变更的时候，动态调整界面的元素位置。

3. 界面元素位置的变更应该做到配置化。

4. 刘海变更的功能单独进行封装，并集成到UI框架中。

5. 细化变更事件与调整逻辑

把上面的点整理清楚，就可以把刘海功能全局化处理。逻辑层只需要配置事件的响应处理方法即可。下面我们再看另外一个例子。

当老的服务器上的玩家活跃程度低下的时候，我们往往会进行合服，合服也是一样，需要一个全局化的处理方式。同样的，我们需要梳理下合服的逻辑。合服是合并数据，有些数据如玩家的等级等记录，是可以直接合并的。而一些排行榜数据就不能直接合并，排行榜数据需要根据2个服的数据重新排列前后顺序。我们需要理清楚合服的规则，以此规则建立配置化合服框架。

除了合服规则，合服的数据不仅仅是玩家数据，也包含各管理器的数据。对不同的数据需要配置不同的规则进行合并。

需要注意的是，我们根据规则所形成的抽象框架，不一定是要继承到系统之中的。像合服这一类的框架，合适的位置是放在工具中。最后我们过一下这个框架的形成逻辑：

确定目标： 玩家数据 管理器数据

确定规则： 直接合并 重排合并

制定框架，配置编程

确定放置位置与执行时机： 工具中，命令行运行

8. 框架哲学

框架里面包含了很多哲学，我们借着laya引擎的一个变更来聊聊框架相关的事情。

laya在某个节点宣布，放弃Canvas API，因为兼容Canvas API会束缚引擎功能。它将毫无顾忌的基于底层的GPU图形API来设计开发引擎。通过这些简单的了解，我们会得出一个结论：底层细节对框架、引擎的设计起到非常大的决定作用。在前面的很多设计中，我们也看到了很多通用的部分，他们似乎和底层细节又有很大的隔离。所以我们可以推断出：一个好的框架既要满足于一定的细节，也需要尽可能的抽象出更高层的通用的部分进行设计。我们可以把它比作一个房子。咋一看我们房子的外形以及朝向等等是可以任意设计的，但是实际上这个房子的建造还是需要实地研究后才能动工。房子的外形无论如何的抽象它都离不开底部地基的支持。所以对于一个框架或者引擎而言，它所处的环境很重要。它是跑在一个APP上或者是跑在一个浏览器上或者是跑在操作系统上，都会对它的框架设计产生很大的区别。

回到刚才我们讨论的案例，laya在刚开始的阶段，引擎的设计主要考虑了Canvas与WebGL的兼容。这种模式下当处于还不支持WebGL的环境中的时候，会自动切换成Canvas的环境，这在早期的手机设备的还不支持WebGL的时候是起到非常重要的作用的。随着手机以及系统的更新换代，现在舍弃Canvas是非常明智的决策，因为环境变了。

当然，无论底层设备如何变化，我们还是应该尽可能的抽象出不变的部分，这是非常重要的。正如我们前面做的那些抽象，无论底层设备发生什么变化，它们都可以很好的应用到新的项目中，仅仅需要调整部分的细节代码。这大概就是框架设计哲学中最吸引人的部分。

第四章

逻辑设计原理

1. 概要

我们这一章是来展示我们是如何去深入去想到或者说总结出上面的模式的。我们可以通过微观的，比如一个定时器的设计来尝试思考。也可以通用一个宏观的，比如如何设计一个跨服系统来尝试去推导出最适合我们的框架。

逻辑设计的原理就是通过对未来需求以及性能要求的预判，来进行对比性思考，得出最符合当前项目的设计方案。它强调的是3个方面：

1．对当前项目的认识

2．对未来需求的预期

3．对比性思考

其中，1、2两点是对项目本身进行了解与认识，3是通用性思考方式。在前面的很多章节中，我们通过反复的对比，推导，比拼各种各样的设计方式，来找出最适合我们的设计方案。这个思考的过程就是逻辑设计的核心原理，它是经得起推敲的。无论外部环境发生什么变化，我们都可以重新应用这样的思考方式，来重构或者拓展项目。

后面的几个小节我们会继续比对各种方案的优劣，继续纠结。在纠结中找到属于自己的最优方案。所有的方案没有绝对正确的实现方式，当外部环境发生变化的时候，有可能就需要调整设计了。

2. 定时器与帧更新

定时器是我们在开发中经常会用到的，比如3秒后干一件事情，大部分就起一个定时器去做。帧更新，通常对应于引擎提供的update函数，是每帧都会执行的函数，在很多引擎都会提供，比如unity就提供了update这样的机制。

定时器能做的事情用update也能做。比如3秒后执行一个事情

```
private nDelayUpdateTipsTime = 0;

private update(nCurTime){

    if(this.nDelayUpdateTipsTime != 0 && this.nDelayUpdateTipsTime <=nCurTime){

        this.nDelayUpdateTipsTime = 0;

        this.updateTips();

    }

}

private delayUpdateTips(){

    let nCurTime = //获取当前时间

    this.nDelayUpdateTipsTime = nCurTime + 3000; //3秒后

}

private updateTips(){

    //更新tips

}
```

可以看到，用update也是能实现同样的效果的。但是很明显的，update的方式处理定时的功能很容易需要引入多一个这样的状态变量，不仅仅如此，在每帧中还多了很多这样的判断，大部分情况下是无用的。但是update也是有优点的，比如说它不需要考虑定时器的管理问题，只需要处理好自己的状态。另外，它们之间最重要的区别是，定时器的编程方式是散的，update的编程方式是集中的。

update对应的很重要的思想是集中化管理，集中化管理很方便去查看代码逻辑。当你看一个类的时候，你直接奔向update函数，在里面看整个逻辑，包含代码的顺序，代码的内容等一目了然。

定时器更加适用于窗口界面等逻辑，在这些界面中，大部分都是被动触发的，不需要一个update一直在跑。很多引擎本身也没有提供组件级别的update的支持，所以这时候使用定时器就很方便，在需要延迟的地方使用。

3.缓存的设计

缓存的提出是为了解决性能问题，用空间换时间。缓存具备的能力与限制：

缓存存放的东西有限，不能所有的东西都放缓存。

缓存应该具备清理功能。

缓存系统应该具备一定的匹配能力。

缓存具备最小保留数量以及预先创建的能力。

举个例子，在某些关卡，会创建大量的怪物，这些怪物会被重复的创建以及销毁。那么当这一波怪物都放到缓存的时候，这时候缓存里面可能有50只怪物，后面再创建30只的时候就可以直接复用。但是后面也可能不再创建了，但是这50只怪物一直在缓存中占据着内存，所以缓存应该慢慢的把这50只怪物的内存释放，直到降到某个最小值。

缓存的匹配能力是指在缓存中通过条件查找到对应对象的能力。需要这个能力是因为我们的缓存里面存放的对象可能是一个简单的实例（不含资源），也可能是实例（含资源）。如果是存放的是有资源的对象，那么因为资源的加载是耗时的，且每个怪物或者其他对象加载的资源往往是不同的。所以当我们有id为1,2,3,4,5只怪物在缓存中的时候且我们要创建5这只怪物的时候，我们并不希望只能拿1来复用而是希望5来复用，拿5复用就可以不需要再次加载资源。

最小保留数量以及预先创建能力都是为了防止第一次创建大量对象的时候产生过多的耗时。这部分的执行应该放在游戏loading的阶段提前创建，并且长期存在于缓存系统中，这样无论什么场合，都不会需要创建过多对象而导致卡顿。这个提前创建对象的个数是根据项目实际情况去配置的，或者根据配置去尝试生成的。

一个简单的示例：

```
class ObjectPool{

private tObjs:Array<any>;

private nMinCount = 0;

    public constructor(){

        this.tObjs = new Array<any>();

    }
```

```
    public pushObj(obj:any):void{
    this.tObjs.push(obj);
}
public popObj(fCondition?:Function):any{
    if(fCondition){
        let tObjs = this.tObjs;
        for(let i = tObjs.length; i >= 0; --i){
            let oCurObj = tObjs[i];
            if(fCondition(oCurObj)){
                let oRetObj = oCurObj;
                tObjs.splice(i, 1);
                return oRteObj;
            }
        }
    }
    if(this.tObjs.length > 0){
        return this.tObjs.pop();
    }else{
        return null;
    }
}
public setMinCount(nMinCount:number){
    this.nMinCount = nMinCount;
```

```
    }

    private nNextRefreshTime = 0;

    public update(nCurTime){

        if(this.nNextRefreshTime < nCurTime){

            this.nNextRefreshTime = nCurTime + 1000;//1秒清理一次

            if(this.tObjs.length > this.nMinCount){

                this.tObjs.pop();//每次清理一个

                return;

            }

        }

    }

}
```

上面的update的是由统一的管理update的地方调用的。上面的例子还没实现预先分配缓存数量的需求。这点我们可以加在setMinCount中，当当前的缓存数量小于最小保留个数时，动态创建若干个对象。当然，这又引发了其他问题，就是我们的缓存对象应该设计成一个模板，这样才能更好的知道要创建对象的类型，这里不继续展开代码。

4. 组件的设计

组件的设计是非常有趣的，组件的出现是为了抽象更加通用化的元素，减少代码复杂的，达到堆积木的效果。列举一些它的优点：

提供统一的ui样式功能，相同的样式功能统一由一个组件实现，比如道具按钮。

窗口本身加载是异步的，异步的东西是不能直接去调用的，因为里面的对象可能还没初始化好。组件需要提供2种方式，一种是和窗口一样的异步加载。另外一种是由机制层提前加载好需要的资源，组件是同步加载，进而调用，大大减少代码复杂度。

组件直接是可以嵌套组件的，几个小的组件可以堆积成一个大的组件。

相似的窗口重复使用类似的组件快速完成界面。

组件反应的思维是，当在处理一个ui或者其他时，如果里面存在可复用的显示对象及功能，那么把它抽象成一个单独的组件。引申的一些组件的设计：

消息怎么路由到组件 ，是从窗口路由还是组件自身监听。比如说有个显示金币的组件，金币需要根据玩家拥有的金币做变更。是在窗口接收到金币变更事件后路由给组件还是组件自行去监听。

组件是无限递归嵌套的吗

组件需要的资源是存放在使用它的窗口（宿主）身上，还是在组件自己身上

还有些更大的问题，组件的这些设计是不是可以跟窗口合并起来，也就是窗口就是组件，窗口本身支持嵌套等。例如，在白鹭引擎的资源层面看，窗口就是一个组件。

我们首先要明确的是组件和窗口一样，都需要一个管理器。但是组件和窗口在设计上有一个核心区别，就是组件是可重复创建的，也就是同时可能存在多个同样的组件，而窗口同时只有一个实例。窗口这样的设计目的是为了让窗口之间更加方便的通信以及避免不必要的代码检测。而组件的设计是基于它原来的需求而出来的。这是核心区别，举例就是一个角色界面，同时只能存在一个，而角色界面里面的道具组件可能有多个。

组件还可以拥有一个能力，快速复用的能力。比如角色界面有很多道具组件，关闭角色界面后，开启活动界面，上面也有很多道具组件，我们希望角色界面的道具这时候应该可以快速”转移”到新的界面上。这也就延伸出了组件的另一个可以考虑的特点：可转移。对应到技术上的实现可以用上一节的缓存来实现。

对于上面的一些设计，我这边给出个人在商业化项目实践的一些设计的选择。首先组件是可嵌套的， 这有利于实现复杂的功能。组件的消息可以自己去监听事件，实现自更

新。它的消息不来自窗口，因为如果来源于窗口，窗口就可能需要广播事件给所有组件，开销比较大。由于组件的嵌套，组件收到后消息后又要广播，消息的派发过程变得一层又一层，引发了极大的效率问题。组件用到的资源是先到父组件去找，没有的话就一层层找上去，直到 宿主UI，最终都没有的话它就需要自行加载。往父亲找是为了加载过的资源不需要重复加载。当然这种资源加载的方式是效率很低的，因为它向上查找的太多次了。我们可以对资源进行管理，使得全局都可以去快速的访问。

5. 资源规划

资源涉及到的东西非常多，资源本身的管理，资源的打包粒度，资源的加载方式，资源的卸载等等。我们在前面简单聊过了加载与卸载。这一节我们聊聊资源的管理与打包粒度等问题。

当资源的数量变多的时候，我们就需要对它进行管理，不然资源看起来会很混乱。

我们先来看2种管理方式：

a) 声音文件夹内含（A的声音， B的声音）

图片文件夹内含（A的图片， B的图片）

模型文件夹（A的模型， B的模型）

b) A文件夹 内含（A的声音， A的图片， A的模型）

B 文件夹 内含（B的声音， B的图片， B的模型）

2种管理方式，第一种是按照类型划分，第二种是按照逻辑功能划分。我们尝试进行一下比较：

a) 如果我们需要加载A的所有用到的资源，那么对于上面的1方式，我们需要去多个文件夹找资源结合来加载。对于方式2而言，是在同一个文件夹中加载不同后缀的文件。这这点上，分不出个优势劣势。

b) 如果有天A相关的资源已经不适合我们的项目的时候，方式2可以很容易的把关联资源删除干净，而方式1很可能会遗漏。方式1不存在类似的问题，因为我们不太可能都不用声音或者图片。

从上面的简单讨论，我们可以感觉到方式2可能更适合我们的项目。我们接着看看方式2的管理。

a) A文件夹 内含（A的声音， A的图片， A的模型）

B 文件夹 内含（B的声音， B的图片， B的模型）

C 文件夹 内含（C的声音， C的图片， C的模型）

＊ 其中A的图片与B的图片相同B的模型与C的模型相同

b) A文件夹 内含（A的声音， A的模型）

B 文件夹 内含（B的声音）

C 文件夹 内含（C的声音， C的图片）

*** 通用文件夹(A的图片B的模型)**

其中A、B的图片位于通用文件夹中， B、C的模型也位于通用文件夹中。

上面的第二种方式较之第一种方式，多出了通用文件夹。很明显，通用文件夹就是放置所有通用资源的地方。把一个东西分为通用非通用是我们经常用到的一个思维方式。把通用的东西抽出来可以减少冗余，不然会导致大量重复的图片或者模型等。但是我们还是要比较一下上面的2种方式：

a) 通用文件夹的方式可以减少冗余。

b) 通用文件夹内的图片不容易被删除，因为不容易知道被几个文件夹引用。而按照对应的功能来划分的话，我们删了就是删了，不用担心资源残留问题。

通用文件夹资源残留的问题我们可以通过工具解决，所以这边我们会建议采用多加一个通用文件夹的方式管理。还没完，通用文件夹中需不需要这样管理：

通用文件夹 =》 图片文件夹 =》 模型文件夹

需要的，有管理一定比没有管理好，这也是一个原则。

通过上面的讨论，我们可以思考一下，在项目之初，我们在美术在划分文件夹的时候，我们就应该尝试去规划出里面的通用部分，减少后期返工的工作量。接下来我们看看资源打包。

资源打包是指把几个小的资源打包成一个大的资源，打包的目的是在于减少带宽，默认情况下我们讨论的这些资源都不是默认打到包体里面的，而是第一次使用或者存在更新的时候才下载的。多个资源消耗的带宽量比打包成一个资源消耗的带宽量大的多。打包的策略有好几种，比如每个文件打一个包，每个文件夹打一个包，几个文件夹打一个包。可以打包成一个zip，或者自定义的格式，或者适用于引擎的结构，这些都是可以的。

我们来看一个例子：

上面的例子描述的是有个ui文件夹，里面有各个模块的文件夹，在它们的文件夹中，又有着多个文件。那么我们可以把ui这个大文件夹打成一个包，也可以把guide，login，task几个文件夹打包，也可以把文件夹里面的文件每个单独打包。

我们需要明白的是，这几种打包方式，在整个项目中，都是会使用到的。但是在ui这个案例，我们只有一种打包方式。我们来对比下这3种打包方式：

a）ui打整包，网络流量是最优的，但是这样第一次就需要下载整个打包，这个是比较费时的，尤其是随着系统越来越多，这个包实际上在后面的版本是越来越大的。还有一个原因是打整包的话，一旦里面有一个文件发生了改变，这个包下次就需要重新下载了，这对更新是很不利

的。

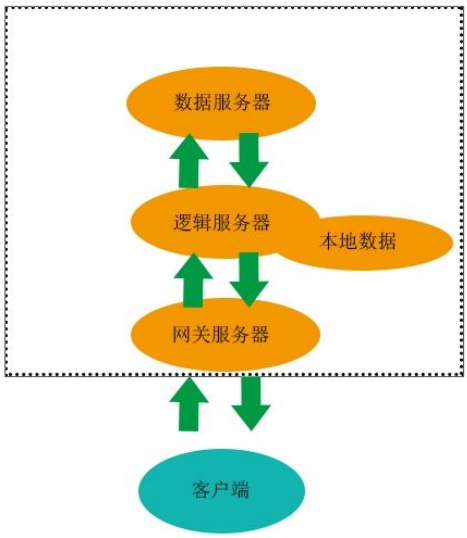
b）每个文件夹打包，流量节约也是可以的。对于更新，它每次需要重下整个文件夹里面的内容。

c）每个文件单独打包，对于流量是不利的。对于更新，它每次只需要更新对于修改的文件，更新量也是小的。

从上面的比较方式来看，很难决定说一定是选用哪个好。因为游戏项目需求是多变的，所有1方式会在后期导致更新量爆炸，所以可以排除掉。那么2,3之间，如何选择？这个抉择我们还是需要去推测一下需求的变更情况。根据经验，一个系统如果稳定了，那么在后面一段时间内都是不会去修改它的界面的，比较多的情况时进行风格的统一，这种情况下通常是一个文件夹内的大部分文件都会多多少少有变化。基于这个经验（不一定是所有项目通用的），我们选择了第二个方式进行打包。

6. 跨服设计漫谈

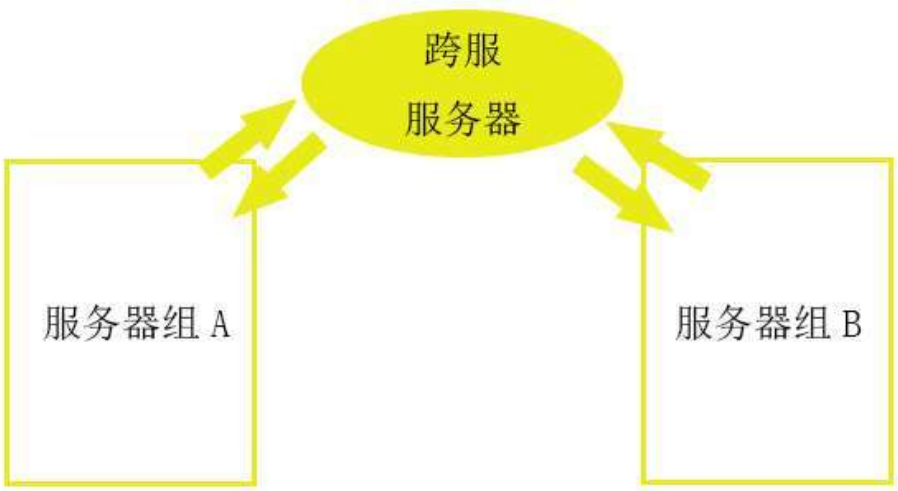
跨服是很多游戏上线后都会加的核心功能，我们这节讲的不是如何实现，而是如何去推导出最正确的实现方式。我们先来看原有的架构：



先简单说一下现有结构。游戏服务器含3个服务器，数据服务器，逻辑服务器和网关服务器。数据服务器存储玩家数据，逻辑服务器存储管理器数据。玩家数据里面是玩家个体的等级，任务等数据。逻辑服务器里面的管理器数据是存放在逻辑服务器的本地数据上，这是为了效率。管理器数据是家族管理器这一类，每个管理器有自己的数据存储。

客户端连上网关后，进行登录逻辑。逻辑服与数据服务器协同完成验证。验证成功后客户端就可以正常的与服务器进行消息往来了。

我们要做的跨服，也就是把不同逻辑服中的玩家对象塞到同一个逻辑服中，使得不同服里面的玩家可以相互可见，一起互动。



我们先整理下看看有哪些思路可以尝试。很直观的一个想法是在原服务器上面下线，在跨服服务器上线。我们先讨论下这种直观做法下我们会遇到的问题。

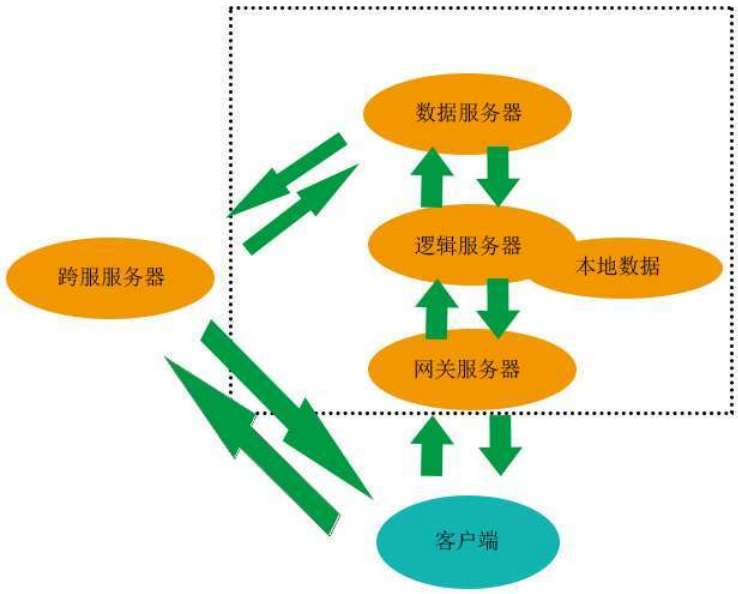
获取某个玩家的在线状态原来是可以直接在原逻辑服获取的，现在原逻辑服判断不存在的情况下还需要去跨服服务器判断。

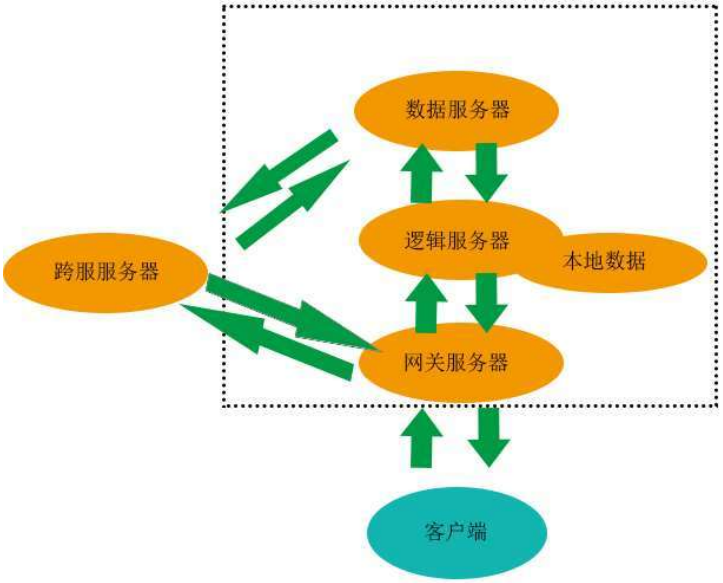
给家族所有发消息，不仅仅要在原逻辑服发送，在跨服服也要发送。而跨服服也需要维护对应的家族关系。

因为管理器的数据在原逻辑服，进入跨服服的玩家的就不能再获得原服的很多信息。

我们现在明白了它的一些问题，后面我们再讨论其他的一些可能性，我们先看看原来的3个服务器之间和跨服服务器应该怎么连接， 这些连接上面就会有好几种情况需要讨论：

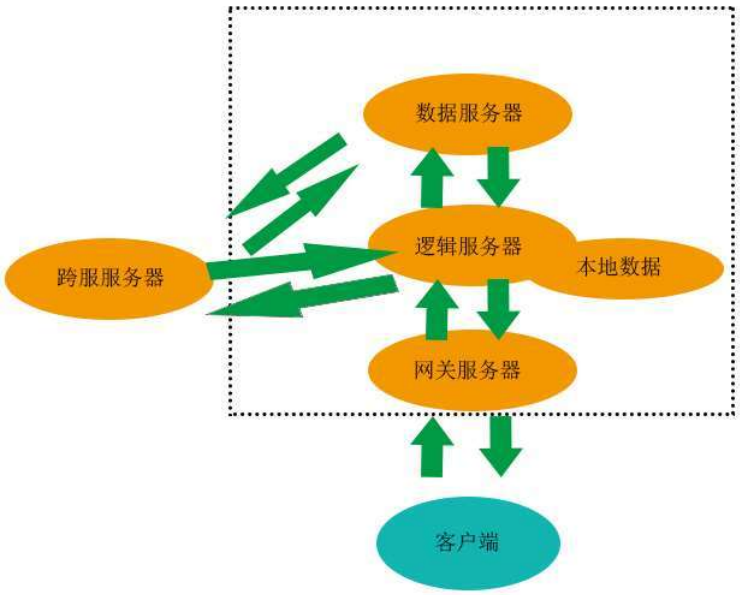
第一种连接方式





第二种连接方式

第三种连接方式



第一种方式是跨服的时候原来的服务器组和玩家客户端不再连接，而是另起一个连接到跨服服务器。这会导致一个问题，就是玩家的客户端必然要走一次断开连接的过程，到时候从跨服服务器退出又得重新去登录，这不符合设想。我们的设想是：玩家在游戏中切换到跨服服务器的时候，对用户是无感知的。有些人可能想维持2个连接，一个连到原服务器组，一个连到新的跨服服务器。这样是存在问题的，因为2个服务器会同时对一个客户端下发消息，造成客户端的数据异常。

第三种方式相对第二种的话，消息的传输链会比较长，每条消息会多经过原来的逻辑服。但是它也有一个优势，就是跨服服与原逻辑服如果有需要数据的同步，尤其是玩家发生跨服行为的时候，这时候玩家的数据传输是比较快的。

整体上，我们采用第二种连接方式可以保证较高的效率。

我们回到架构问题上，继续讨论跨服的时候将玩家从原来的逻辑服移动到跨服服务器的这个决策上。我们前面提到它有若干个问题：原服状态丢失，消息广播以及管理器数据丢失。其他的包括顶号，断线重连等问题，这些问题都处于一个我们可以忍，可以憋屈的解决，也不是不能完成的情况。也就是说，这个架构在目前我们很难直接去推翻它，你可以用这个架构去实现跨服功能，但是会比较痛苦。

随着一系列的痛苦的发生，我们才得以搞清楚一个情况：未来新增的系统也因为跨服，导致在编写代码的过程中需要去考虑是否处于跨服状态以编写各种适应的代码。这个过程是无止境持续的，也就是说我们现在的一个系统无止境的影响了未来的系统，打破了基础的低耦合逻辑，这直接动摇了跨服架构的设计。

因为有了架构的冲突，我们需要改变架构，无论之前在原架构上用了多少的思路处理问题，都需要改掉。同时我们需要研究一个新的架构来支撑这些需求。

既然我们没有办法把一个玩家完全从原逻辑服移到跨服服上，但是我们确实需要玩家在跨服服上进行互动，那么我们只能考虑玩家同时存在于原逻辑服和跨服服上，也就是他在2个服务器上都有一个实体，也有对应的连接。我们前面说过，当2个服务器都有连接的时候，它们都可能给客户端发数据，这样很容易产生异常情况。那么我们可以考虑所以的逻辑，数据都跑在跨服服上，但是这样和完全将玩家移动到跨服服就是没有区别了。从这个角度，我们可以考虑部分逻辑跑在跨服服，部分跑在原逻辑服。根据跨服本身需要的互动，我们可以剥离出玩家实体，玩家移动，玩家战斗等逻辑到跨服服。其他部分，如购买道具、升级等玩家模块逻辑都放到原逻辑服处理。通过上面的划分归纳总结，我们可以总结出跨服过程中，把游戏核心交互，战斗等逻辑放到跨服服的对象上实现。把游戏其他的逻辑放到原逻辑服。这样有一个明显的好处是： 非核心逻辑是可以直接接触管理器数据的。

到这个地方，思路方面的探索已经清晰了一些。我们转而思考一点细节来看看是否存在其他问题。在对一个框架的探索过程中，需要从思维逻辑到细节不断切换，才能更好的理清楚其中的思路。我们刚刚把一个整体逻辑拆分为核心逻辑与其他逻辑，分别跑在不同的服务器上。一个整体分成两个部分，在外面又需要表现的和一个整体一致，那么我们就需要解决两个部分之间的数据同步问题。我们来举几个例子：

a) 玩家等级升级（非核心层）导致玩家战力属性提升（核心层）

b) 玩家杀怪（核心层）导致获得经验（非核心层）

上面2个例子说明了他们之间是存在交互的，而且这个交互是双向的。我们再留意一下，怪物杀怪需要计算的经验，依赖于玩家的等级（非核心层）。所以说，在跨服服核心层的一个行为，不仅依赖于非核心层的数据，也影响着非核心层的数据。基于对称，我们也思考下非核心层数据是否也存在这样的关系。非核心层数据依赖于核心层的位置，比如说探测器的判断（非核心层）依赖于玩家坐标（核心层）。同时，非核心层也对核心层产生影响，比如玩家升级（非核心层）使得玩家属性（核心层）产生变更。

我们继续讨论下依赖与同步之间意味着什么。首先看简单理解的，A数据同步到B就是当A发生数据变化的时候，通过网络将数据同步给B。A依赖于B意味着需要同步数据给A，这样A使用数据做判断才是有效的。所以依赖也是一种同步，同步的核心就落到的同步的数据上。

我们开始讨论跨服需要同步的数据上。当我们把场景限定在跨服的时候，事实上是不存在非核心层依赖核心层的。我们在跨服的2个服务器进行职责划分的时候，其实已经把它们直接的耦合部分解开了。那么我们下来分别讨论下互相同步的数据。

逻辑服（非核心层）当发生升级等的时候，需要将属性同步给跨服（核心服）。

所以属性，战斗相关的技能等等一般由C++实现的都需要同步给跨服。在跨服上，杀怪经验等依赖于等级。所以，等级这样的一些属性也要同步给跨服。跨服杀怪产生经验，需要同步经验增加操作给原逻辑服。注意这边是同步操作而不是同步数据。数据只能在它所在的服务器进行修改。比如说经验，是属于非核心层，位于逻辑服。那么它只能在逻辑服修改，跨服服只能同步操作。如果跨服服也同步经验，那么在网络传输过程中，可能导致经验出现计算错误。同步是有网络延迟的，这是导致这个错误发生的根本原因。

经过上面的分析，我们来看看新的架构做了什么，是否还存在问题。新的架构下逻辑服负责非核心层的逻辑，跨服服负责核心层的逻辑。非核心层逻辑又可以使用管理器数据了，跨服服完成核心战斗等逻辑，通过数据同步解决了2边数据的一致性问题。对比之前的框架，我们现在的工作量主要变成了同步战斗层的数据，这个数据也是不少的。但是，对于长期的需求变更而言，战斗逻辑是相对不变的，而其他非核心逻辑是一直处于变化之中的。所以在现在的架构下，我们只需要重点解决核心逻辑。而未来的需求变更，我们只要处理好额外产生的依赖和同步，不用过多的关注跨服的存在，这大大的保证了新系统的开发效率。

我们最终采用了镜像式的跨服架构来完成我们的需求。我们的逻辑思想之路不能就此停止。我们考虑一个情况：如果我们早早就知道这个需求，我们设计的架构应该是怎么样的？对于之前废弃的方案，如果我们把管理器数据都存放在数据服务器上，我们第一种方案是不是也是可行的方案？那么两种架构的比拼最终就进入到了细节实现难易程度的比拼，又值得更深入的探讨了，我们这里就不继续展开了。

这种跨服细节的实现也是有很大内容的，因为本书不是讲细节的。所以我们只罗列一些思考点：

1) 逻辑服不小心操作了核心层数据导致同步给客户端应该怎么办？

2) 跨服服不小心操作了非核心层数据导致同步给客户端应该怎么办？

3) 跨服服是否可以给客户端推送一些无关数据的消息？

4) 跨服服和原逻辑服同步的时候协议是怎么制定的？

5) 跨服服需要离线发一封邮件，但是玩家已经下线，找不到对应的逻辑服怎么办？

6) 动态跨服如何不停服完成？

7) 跨服中的断线重连如何处理？

8) 跨服服的代码和逻辑服的代码用同一套还是各写一套？

7. 分线漫谈

分线，是将玩家划分到不同的频道中，不同频道的玩家互不可见，且不会互相同步消息。分线在程序方面主要用于减少网络包，在策划层面会有一些其他的应用。分线是基于场景的，我们的可见性，以及消息同步默认以场景为单位。

在不分线的情况下，玩家之间是默认互相可见的，且可以互相同步消息。这种情况我们给它个定义，为0线。0线的对象与其他任何分线的对象之间都互相可见。接下来我们定义总的分线数量，这是个配置值或者计算出来的值，根据服务器导入的玩家人数来决定。如果同时导入了大量的玩家进入服务器，就会存在大量的网络包，那么这时候我们就要开启分线。假设我们最多能支持1000个玩家，那么同时导入4000个玩家的时候我们就要把总的分线数量定义为4（4000/1000）以及4以上。接着上面的例子，我们就假设我们需要分4线。我们来看看玩家是怎么分配到各个线的。有2种分配方式：

a）第一个玩家进1线，第二个进2线，第三个3线，第四个4线，第五个1线。

b）前1000个玩家进1线，后1000个玩家进2线，以此类推。

第一种分配方式下，来了1000人，每条线250人，没有那么热闹。第二种分配方式，来了1000人，都在一线，网络需要同步的量大了。

我们再来聊一下分线方式。

全局分线，设定好分线规则，对不同的场景决定是否启用该规则场景分线 对不同的场景应用不同的分线规则

场景分线可以说是全局分线的一种细化，来自于策划的需求。策划的需求可能在某个等级，将大量的玩家汇聚到一个场景中，其他时候又分散开来。这种情况下，如果我们为了这一个场景去提高分线数量，会导致其他场景的人数变少，进而导致场景凄清的感觉。

我们先以全局分线的方式进行进一步的讨论。我们来看看几个问题：

a）不同分线的玩家组队的时候，是否需要调整分线？

b）位于A场景分线1的玩家，从A场景进入B场景，他的分线是否需要发生变化？

c）场景A中的分线1的玩家与分线2的玩家都在攻击某只分线0的怪，但是互相看不到对方，经常看到怪物突然死掉（被其他分线玩家杀死），体验不好，如何解决？

我们讨论下上面的问题。不同分线下玩家为了组队，队友之间应该是可见的。所以我们需要把它们划分到同一个线。我们可以强制让他们都去1线，这样会暂时导致1线玩家数目增大。当他们退出队伍后，会重新进行分配。位于场景A中的分线1玩家，到了其他场景，

他的分线应该发生变化。这是因为分线是基于场景的，进了新场景，就需要跟着新场景的人数而变化分线。

第三个问题是比较大一点的，在分线过程中，有些怪物是按照分线的数量来创建的，有些是放到0线去抢的。按照分线刷出的怪只有特定分线的玩家才能看到，这样每个分线的玩家都能获得一致的体验。而一些特定的怪，如大boss等，我们限定了数量，就只能放到0线给大家去抢。所以，我们需要策划去思考清楚哪些怪物按照哪种方式去创建。在关卡编辑器或者配置表中，我们需要添加相应的支持。

第五章

细节与其他

1. 代码管理

本节介绍2种代码管理模式

大型项目中会遇到一种代码管理模式，基于分层的思想。底层代码会在一个代码库中，项目上层代码会在一个代码库，项目工具代码会在一个代码库。三个代码分别检出构成最终的项目。

这个管理模式是很好的，主要解决了不同人的代码权限问题。根据项目的初中高级别程序，开放不同的代码。但是这个模式有一个问题，就是不利于复制。我们需要分别把几个代码库检出到对应的文件夹，提交的时候分别提交。当然，这些我们都可以使用工具去做这些事情。

另外一种模式是在小的项目中，复制就具备了更加的重要性。所有的代码放在了一个库中，这个库不是按照代码分层来分的，而是以项目格式来建立的。这个时候，构建新项目变成了复制基础库，在基础库上添加需要的东西。

上面的2种模式，我们可以梳理出的结论是：越复杂的东西，越需要控制风险。越简单的东西，越需要做到快速复制。

2. 检测与转换

在项目里面，程序经常会出现一个情况。策划配置配错了，导致程序脚本崩溃了。这在项目里面是个灰色地带，没有人愿意为这个情况去负责。为了杜绝这个情况，我们会引入环节，配置检测。

配置检测有2个作用：

减少代码负担，在代码中不需要去判断获取的配置内容是否是需要的类型，符合需要的规范。

及时找到配置的问题，保证配置的正确性。

下面主要谈检测的时机，如何检测。

检测的时机放在策划发布配置的时候。策划发布配置是指策划在修改配置文件后，生成程序需要的文件并提交到服务器上的时候。不直接放在策划修改配置表的时候是因为会影响到配置的效率，需要单独在csv或者xml上加入项目级别的限定，这样不方便程序去做限制且无法拓展。这个发布配置的环节，使用和项目运行时一致的代码进行检测，可以保证配置是完全符合项目需要的。

配置检测主要是检测表与表之间的关联性，检测是否存在重复id行，检测值的范围是否正确。这些都可以使用之前章节提到的配置化编程实现。

这边还有一个重要步骤，就是在检测的同时进行转换。因为策划需要的表格式与程序需要的表格式可能是不同的，这主要取决于效率的因素。通过在这个步骤进行配置转换，使得一些需要在运行时执行的配置合并等操作，在这个环节就得以完成。

3.缓存更新

我们在做一个h5游戏的时候，因为浏览器会缓存资源到本地，经常会导致我们希望更新的东西没有正常被更新。同样，我们在自己设计App游戏的时候，比如A资源已经被下载了，那么下次访问A资源的时候就不需要再向服务器请求了，这种支持极大的减少了带宽。所以，我们也一样会实现一个类似于浏览器一样的缓存系统。那么当A资源被修改了之后，我们也希望下次加载A资源的时候能下载到新的资源，而不是沿用老的资源。

合理的更新方式是我们需要区别资源修改（包括代码修改）这个行为。所有的修改都会导致文件变更，这就给了我们机会去”计算”这个修改。我们来看下更广阔的场景：

A资源的文件名为bg.jpg，代码B文件中直接引用了bg.jpg。

我们希望B文件能带有唯一的标识，我们可以用它的文件内容来标识它。当文件内容变化的时候，它也就有了新的标识。那么B文件名字可以变成bg_XXX.jpg(XXX为文件内容)，这样就可以区别每一次的文件修改了，因为文件内容一变，我们也会把对应的文件名改变。再把对应引用的地方修改，就不会导致更新问题了。修改后的场景为：

A资源的文件名为bg_XXX.jpg，代码B文件中直接引用了bg_XXX.jpg。

这个改动实际上是存在一些问题的：

bg文件使用文件内容来定义文件名会导致文件名过长，超过系统的文件名限制。

每次修改bg.jpg都会导致所有的引用都需要修改，我们可以写一个工具，但是这也会影响开发效率。

针对上面2个问题，我们可以将文件内容通过某个算法，比如CRC32生成一个32位字符串，截取字符串的后面几位，与bg进行拼接，拼接后的字符串为bg_1C6D35.jpg(1C6D35是截取后的），这样第一个问题就解决了。因为每次都需要修改引用，这个时候我们可以引入中间层来解决这个问题。使用res1 来定义 bg.jpg的别名，所有使用bg.jpg的地方都改为使用 res1来访问。而每次修改bg.jpg的别名的时候，我们用工具来修改 res1和bg.jpg的映射关系即可，其他引用的地方都不变。最终的映射变成了 res1 =>bg_1C6D35.jpg，这解决了我们第二个问题。最后我们来聊时机问题，也就是这个映射的修改我们应该在什么时候进行。每次改变文件内容的时候进行会影响开发效率，所以我们把这个操作放在发布的时候进行。

4. 异常处理

这一节我们讨论异常处理。异常是我们在写代码过程中经常遇到的，我们把一个脚本的崩溃也称作发生异常，这是因为脚本的崩溃会导致后续代码中断执行，导致不可预料到的后果。针对异常我们经常用的方式是去捕获，捕获后清理。我们来看一个例子，服务器玩家执行update。

```
for(let i = this.tPlayer.length()-1; i >=0; --i){
        let oPlayer = this.tPlayer[i];
        oPlayer.update();
    }
```

代码的意思就是遍历所有的玩家，执行他们的update函数。但是player的update是可能发生脚本崩溃，为了一个玩家的崩溃不影响到另外的玩家，我们需要加上异常捕获：

```
for(let i = this.tPlayer.length()-1; i >=0; --i){
        try{
            let oPlayer = this.tPlayer[i];
            oPlayer.update();
        }catch{
            this.tPlayer.removeAt(i);
        }
    }
```

在每个玩家执行update的外部用try包裹住，当发生异常的时候，把玩家移除。关于try catch的效率问题见下面的一个帖子

https：//www.zhihu.com/question/29459586

我们这里不去讨论try catch本身是否会存在性能损耗，我们假设try catch是存在损耗的，那么上面的循环就会导致过高的损耗发生，会花费一定的时间，这是无法接受的。接着我们来优化一下，我们还是不能放弃异常处理，但也希望能更加有效率。

```
let nIndex = this.tPlayer.length()-1;
```

```
        while(true){
            try{
                let oPlayer = this.tPlayer[nIndex];
                oPlayer.update();
            }catch{
                this.tPlayer.remove(nIndex);
            }
            if(--nIndex < 0){
                break;
            }
        }
```

这是优化过的，这样的实现不仅仅可以保证异常控制在一定范围内，也可以保证效率。

下面我们来看一个简单的异常框架：

```
let nIndex = 0;
    try{
        //干第一件事
        nIndex = 1;
        //干第二件事
        nIndex = 2;
        //干第三件事
        nIndex = 3;
    }catch{
        swtich(nIndex){
```

```
            case 0:
                break;
            case 1:
                break;
            case 2:
                break;
        }
    }
```

这段代码主要表达的是，我们可以根据异常的不同阶段，进行不同的清理。比如到第一阶段完成后发生异常需要清理什么，第二阶段后需要清理什么。拿剧情系统举例，如果在某个剧情发生了异常，那么需要把前面剧情创建的对象全部清理掉。

总结下异常的处理，我们需要先划分最小的异常范围，然后在进行少量的异常捕获，在捕获后需要把异常的对象清理掉。这就是整个异常处理的过程。这里需要掌握的是核心的处理方式，就是我们怎么样可安全的处理好一个不确定性能的代码。有时候我们不需要了解到底是否这个代码有耗时什么的，只要优雅的使用代码，像前面的章节说的那样，用我们的思维习惯减少需要了解的知识量，提高代码的稳定程度。

5. 适配这件小事

适配，是所有游戏都会遇到的事情。无论是app还是h5游戏，在不同的设备屏幕，不同的浏览器都会遇到适配问题。

对于很多引擎，它们都会提供不同的适配方式。比如白鹭引擎里面，提供了：

1. SHOW_ALL 等比缩放适配你的窗口。比如游戏设定是480x800的。如果页面是640x700，则上下满屏，两边会出现黑边。如果是640x1280，则两边满屏，下边会有黑边。

2. NO_BORDER 会不按照比例，按照页面的宽高暴力充满整个窗口。

3. 其他适配方式。

有些引擎提供了很多的适配方式，有些引擎不提供任何适配，那么我们是不是有通用的适配法则呢？下面我们提供一下解决方法：

1. 无论页面大小是怎么样的，都需要在加载的时候将设计分辨率设置为与页面一致。

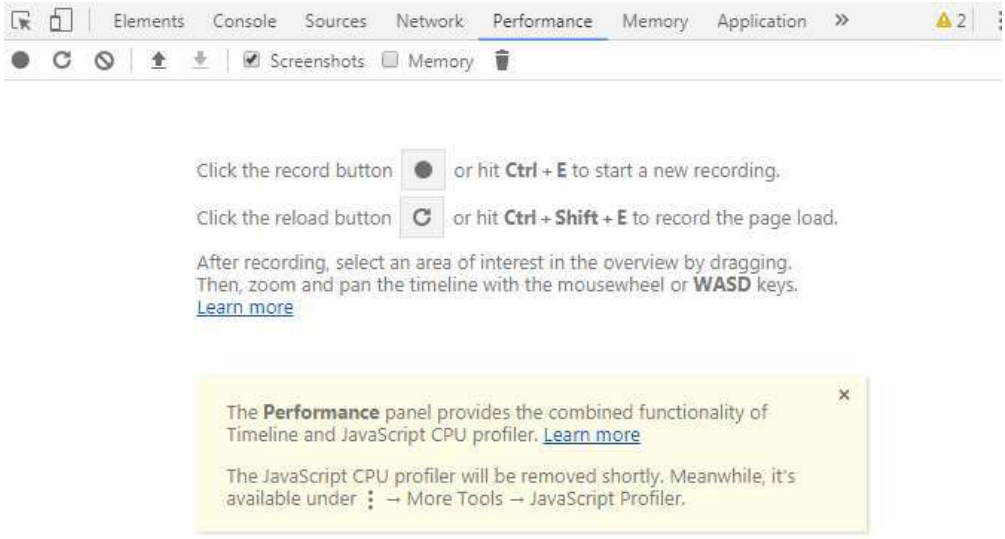
2. 将游戏内的UI与场景划分，UI采用布局管理。场景地图放置在中间，并且保证足够大。

在这种处理下，不同的屏幕看到的UI的位置都是相对的，看到的场景的区域会不同。当然，我们首先要保证场景与UI是不存在视觉关联的，也就是说场景和UI 换了哪一处都可以正常显示。背景很花哨，UI只能放在某个地方才能显得比较和谐，这种场景就是不行的。其次我们要注意到，我们提供的适配方法一定是建立在布局的基础上。如果一个引擎或者我们自己没有实现对应的布局系统，那么通用的适配也就无从谈起了。布局是适配的根，就像相对速度和速度之间的关系一样。

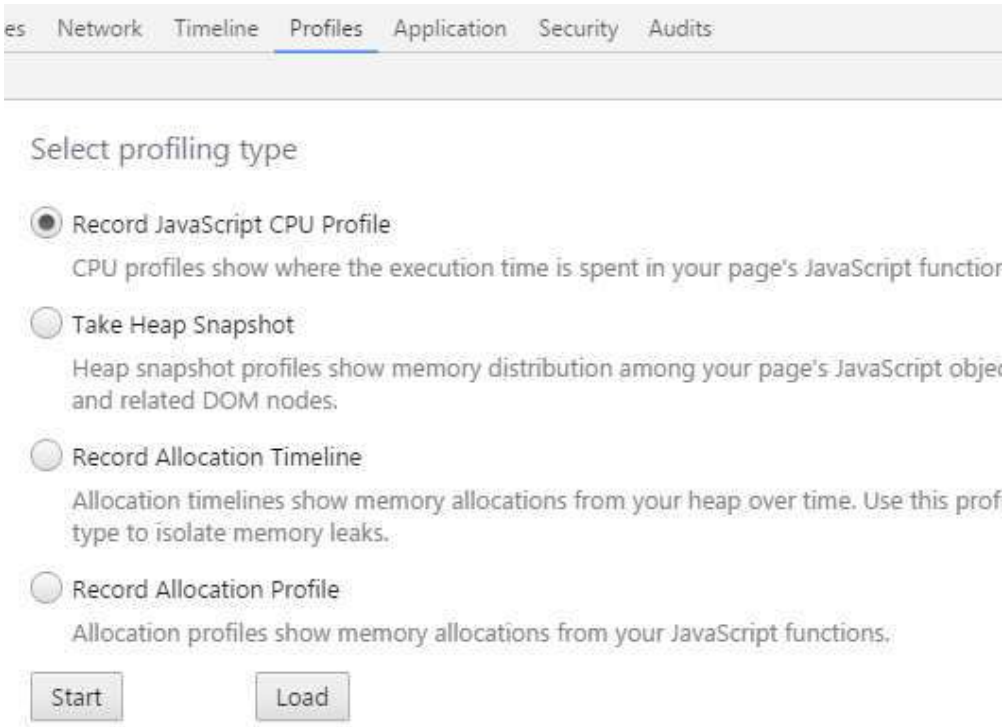
6. 性能分析

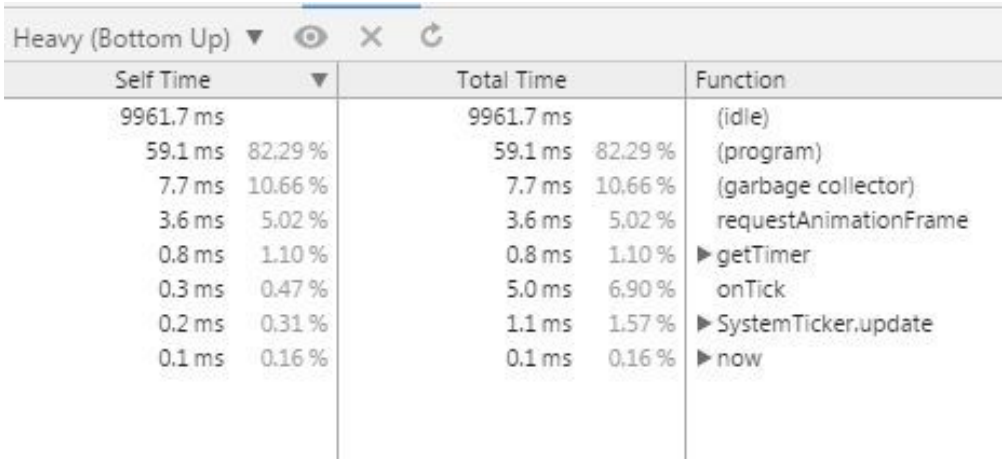
性能分析，常被称为profile。profile是个常用的工具，主要用来分析项目在一段时间内的运行情况，包括CPU、GP、内存的使用情况等。同时，它也经常被用来分析游戏中的异常情况，比如卡顿等。

我们来看下如何对js代码进行profile，以chrome浏览器为例。chrome浏览器在控制台中提供了一个performance的功能，用来对整个项目进行分析（如下图）。



其他版本的chrome里面会看到profiles这样的选项（如下图），也是类似的。





我们尝试对CPU的情况进行profile，我们会得到下面的情况（如下图）：

上面列出了在这段profile时间内的函数的耗时情况，我们可以选择其中耗时最多的函数细化分析，看看到底是什么地方导致的高的耗时。高的耗时可能是代码写的不好也可能是调用了比较耗时的api。我们做深度性能优化的时候可以经常用类似的工具来帮助我们。

实际上，对于我们项目中的逻辑，我们也是需要建立各种各样的profile来监视项目的情况。我们要注意到，profile不仅仅是一系列的工具，也是一种思想。它代表着对整个项目性能情况的把握，可以有效的规避技术bug。

我们在项目中可能会创建大量的玩家以及大量的怪物。那么我们需要监测：

1. 每帧玩家，怪物各自的执行时间。

2. 玩家，怪物的实时数量。

做这些监测可以使得当服务器发生以下情况的时候，我们能够通过输出或者dump下服务器的内存，来快速分析出异常原因。

1. 响应时间异常，CPU高涨

2. 内存一直升，没有降

举一反三的思考，我们还可以监测处理网络消息的时间，定时器执行的时间，管理器内对象的数目等等。这些种种情况，我们可以通过监测，也可以在适当的时候输出它们的信息分析。

至于如何去实现这样的一个系统，我们需要做好2点：

1. 实现好时间上的分析。比如对应一个函数，我们在函数开头记录时间，在结束也记录时间，从而得出函数的具体执行时间。

2. 实现好状态的记录。比如记录怪物的数量，或者记录怪物分配的内存。

对于逻辑层面的profile，更多的可以记录数量而不是底层的内存。不仅仅可以记录数量，还可以记录状态，或者命中的次数等。任何有帮助于分析的东西都可以被记录。

从大量的项目经验中，我们可以知道的是，大部分的问题发生在逻辑层面。所以我们要针对逻辑部分做好充足的监测。思考问题先看看是不是逻辑问题再往底层思考，这样会比较快的找到问题。还记得在第一章的code review代码中的update吗？曾经一个版本的时候就发生了bug。我们调试的时候依然是先看看自己的使用方式是否出现问题，然后再看底层的代码。通过review技巧去挖掘到相关问题可能发生的地方。

这边我们要注意到，profile并不是常用的调试手段，经常我们还是使用前面的调试方法。我们使用profile主要是在预防以及处理疑难杂症上，不要过早的使用profile。

第六章

商业环境问题

1.线上问题的处理

当我们的项目上线后，很容易出现很多我们在内部没有发现的问题，有些甚至可能会低级，但是它就这么发生了。有些很难重现，我们需要想尽办法去捕捉它们。

我们来看一个例子：

项目正常上线几日后，运营协同策划技术决定进行一次更新。更新过程很顺利，该修复的bug在外网都正常验证成功了，但是这时测试发现结婚的排行榜上没有显示任何的结婚数据了，排行榜原来是以亲密度高低进行排序的。在内网测试后，我们发现结婚的排行榜系统没有保存数据到磁盘中。这个问题是非常低级的，从技术到测试都不该发生的，但是确确实实发生了。这个问题刚发生的时候，往往考虑的方式是想办法从他们的行为日志里面去挖掘信息。那么这个做法需要游戏开发的程序与后台组（负责日志模块对接）一起协同去从海量日志里面去还原。这个亲密度的来源是很多不同的行为，比如组队会产生亲密度，聊天也会产生亲密度。来源非常多，初步估计这个问题处理好要半天的时间。

最终这个问题的处理并不是像上面这样处理的。因为在更新的前一天，有玩家在群里面截图了这个排行榜的信息，我们通过这个截图快速的重新构建了数据，完成了这个事故的修复。

这个例子告诉我们2个点：

1）遇到线上事故不要慌，先去收集好有用的信息再处理。

2）在项目运行的过程中，我们要收集留意对项目有用的信息，这些信息在关键时候可能会起到重要作用。

我们再看一个例子：

有一天夜里，电话响起来了。发现服务器出现了一个情况，也就是基础章节提到的不对称异常。日志输出如下：

```
OnGateConnect 1 0

OnGateConnect 2 1

OnGateDiscconnect 2

OnWorldDiscconnect
```

这个案例通过上面的思维方式，我们可以得出存在一个不正常的连接，导致了逻辑上没有兼顾到。但是在运维层面，并没有工具可以知道是哪里发起的异常连接。这个时候，采用的方式是修改了服务器的端口为其他端口，这样就不会收到那个异常连接了。然后在

搭建一个测试环境来复现这个情况，最后解决。这个例子告诉我们，如果我们没有办法快速的找到问题，那么我们可以使用一些规避方式来延长我们的处理时间，避免卡在一时导致玩家一直无法正常游戏。

再看一个例子：

后台的留存数据与渠道方面的数据不一致。经过研究发现，后台在处理过程中把一些消息给阻挡掉了，导致没有正常进入分析。这个问题还是比较好办的，因为游戏这边还有日志信息。只要有这些原始的日志数据在，就不担心丢失的问题。这边也要注意下数据如何还原。数据不能通过游戏方重新发送给后台，因为这样会出现时序性的问题，后台已经处理也一些数据，在处理前面的数据，就会导致处理的顺序不一致。所以只能是后台使用游戏方的数据进行数据还原。

这个案例主要说的就是，如果我们在设计之初，在游戏方本身就做好了数据或者日志的保存备份，那么我们就可以应对外界各种异常情况。

最后看一个例子：

某台windows服务器上的游戏服务器的内存持续飙高，运维紧急的dump了内存， dump后生成了一个dump文件，非常大。服务器的内存已经无法对它进行调试了。这个问题不知道什么时候就会触发，每次发现的时候内存都已经是这样爆了，最终都无法直接调试。

程序方面很紧张，手忙脚乱的准备下载到本地后分析dump文件。问题出现了，从外网下载到内网，这样一个大的文件需要花6小时左右，这基本无法等待。最终这个问题的处理是在外网搭建了一个调试环境，将这个dump从事故的机子直接移动到调试机子上进行分析解决。

这个案例主要是告诉我们，遇到紧急的问题，需要灵活处理，优化整个处理时间。保持对问题的灵活思考是关键。

上面4个例子总体阐述了遇到问题的一些解决方案，保持不慌不乱的心态，灵活的处理问题是最重要的。

2. 程序负责人的关注

程序负责人在项目中一般是位于第一环和最后一环。第一环是确定方案，最后一环是救火。本节介绍一下程序负责人需要关注的东西。

微观：

配置的定义

存储的数据结构定义

通信协议的定义

服务器，客户端模块通用部分的抽离客户端界面公共部分抽离

细节难点的处理

宏观：

内存的稳定

CPU的稳定

网络包量的稳定

程序本身的稳定

兼容性问题

综合：

资源的规划

设计的决策

代码规范的确立

后台（php）交互方式制定

在整个项目的执行的过程中，需要预先研究出合适方案，确定设计的思想。从设计中的抽象出通用的跨项目的部分，以及项目内通用的部分以及特殊的部分。在执行过程中，确保数据结构与配置是吻合需求且是尽可能匹配未来需求变更的。在代码方面，需要经常review以便代码风格的统一，且代码是可持续修改的而不是引发灾难的。优化也是很重要的，

性能上面的优化。时间，空间上面的占用都不能消耗过多。这些东西能尽可能的保证随着项目越发庞大，项目不至于被拖垮。

项目到了后期，实际上是希望随着经验积累，需求可以做的越简单，系统可以越稳定。但是在项目的迭代过程中，程序本身会进进出出，需求改来改去，系统中的不合理的地方只会越堆越大。到后期，1个简单的需求可能就因为历史原因而需要花费较多的开发时间。所以在整个执行过程，不仅仅需要一个好的流程，也需要经常去迭代当前系统的代码，去优化，去做团队技能升级。

3. 商业限制杂谈

在一个商业项目中，会遇到各种各样的限制。商业项目需要作出最正确的策略，从某一方面而言，商业项目就是一个复杂又精妙的策略体。

下面列举一些商业项目的要求：

启动速度快

登录速度快

包体小

低端安卓机器很少卡顿，跑的流畅

很少闪退

客户端可以自动热更新，服务器可以热更新配置且不停服修复大部分bug

流量使用少

发热少

服务器稳健，不允许有回档

兼容性高

下面我们聊一聊每一个点的要点：

1）启动速度快我们可以做几个事：

a）减少启动要加载或者说处理的事情。

b）使用多线程加速。

2）登录速度快：

a）登录过程协议交互少

b）验证速度快

3）包体小：

a）包体内只含有部分重要的东西

b）大部分资源都按需下载

4）低端安卓机器很少卡顿，跑的流畅

a）减少gc以及耗时操作

b）降低低端机的表现

5）很少闪退

a）留意多线程的使用

b）留意native层的代码问题

6）客户端可以自动热更新，服务器可以热更新配置且不停服修复大部分bug

a）充分的脚本支持

b）相应的命令支持

7）流量使用少

a）同步时尽量只同步必要信息

b）同步数据可以合并后再同步

c）减少不必要的同步

8）发热少

a）降低CPU的消耗

b）降低GPU的消耗

9）服务器稳健，不允许有回档

a）捕获游戏异常，做好异常处理

b）即使出现回档，也需要让玩家减少损失

10）兼容性高

a）尽量使用统一的数据类型

b）优雅的使用代码（见基础章节）

我们选取其中的一两点进行稍微细一点的讨论。

第一点我们讨论包体；

第二点我们讨论回档问题。

包体的大小根据不同的游戏类型有不同的限制，这是因为包体的大小对于推广会有很大影响，所以包体只能包含有限的资源，包体越小越有利于推广。游戏内含的内容是非常多，大到几百兆甚至是几G。包体资源打包涉及到的问题本质是优先级问题，越重要的，越通用的资源应该被放到包体中。我们来梳理下优先级：

启动需要的加载资源，用于加载其他资源

程序脚本

配置文件

通用的ui资源

通用的特效资源

通用的声音资源

登录用到的资源

从低等级到高等级用到的通用资源从低等级到高等级用到的非通用资源

接着聊回档问题。在游戏服务器的运行过程中，可能会因为各种异常情况导致游戏崩溃。服务器意外崩溃，那么玩家最近的数据变更就会丢失，因为玩家的数据不是时时刻刻保存的，一般是一阵子保存一次。既然崩溃无法完全避免，那么我们需要做一些措施来补救。我们发现，如果崩溃的过程中玩家升了级，那么崩溃后他等级被还原了，那么这就是很不好的体验。崩溃前玩家可能发生了充值，服务器重启后，玩家买的道具也没了，这也是不好的体验。所以，我们可以发现，我们需要去挽救一些情况，以使得即使发生了服务器崩溃，玩家能没有怨言。解决这个问题有2个思路：

a）在服务器重启后去给玩家补贴

b）在关键数据变更的时候强制保存

方式1的话不好评估需要补贴的东西，而且通过补贴不一定能平玩家的怨气。方式2的话可以避免服务器崩溃玩家等级变更或者金币减少的情况。可以得出方式2的处理方法会

优于方式1。那么针对方式2的处理方式，我们需要整理出哪些数据是重要的。我们可以尝试整理下：

a）等级发生变化的时候

b）主线任务发生变化的时候

c）发生充值的时候

d）vip等级发生变更的时候

还有很多数据，只要是比较重要的，我们都可以强制保存。当然，我们在保存的时候要做到即使有多个触发条件同时满足，我们也只能触发一次强制保存。在保存方面，我们还可以有一些其他的策略，比如没有那么重要的数据，我们可以在它的值变更若干次的时候强制产生保存。用变更次数触发保存来定义一个数据的重要性，是一个很好的量化手段。重要的数据，就是变更1次触发。不重要的数据，可以变更多次触发或者等到定时保存。

4. 妥协与策略的应用

在开发过程中，会遇到很多需要妥协的地方。有些需求很苛刻，如果我们直接去实现的话可能会遭遇性能障碍。有些情况是必然要发生的，屏蔽或者绕过可能会导致体验的大幅度下降。这时候我们就需要妥协，想其他办法来解决问题，这些方法我们通常称为策略。

接下来我们看几个例子来感受下策略的魅力。

第一个例子：

玩家切换场景的时候，出现的点的旁边有很多其他玩家。这时候我们会发现游戏在进入这个场景的时候会卡顿，而进入玩家少的场景就不会卡顿了。经过一番排查，我们发现同时加载很多玩家会导致卡顿。我们对于这个问题也不能说不加载，所以我们可以慢慢加载。

所谓的慢慢加载就是指把这整个加载过程放到好几次的处理里面去执行。这边说的次可以是帧，或者一定的时间间隔。比如说我们每隔1秒加载一些玩家，再隔1秒再加载一些，这样就可以解决加载卡顿的影响。

我们可以优化一下加载方面的体验，比如说我们对要加载的玩家按照距离排序，先加载近的，再加载远的。这样的话我们就可以慢慢的看到其他玩家从近到远被加载出来了，而不至于看到若干的玩家无序随机的出现。

对于加载这块，也存在的很多策略。我们可以分次加载一定数量的玩家，也可以按照固定时间来加载玩家。对于CPU时间上面的控制，我们通常会建议按照固定时间来加载，这样不至于突发情况影响了这一帧的时间导致卡顿。

上面的问题的处理里面有很多策略的东西，从加载本身，到体验优化，到加载的具体策略，这些都是策略层面的东西。

第二个例子：

游戏场景里面存在很多的光源，这些光源都会对游戏内的建筑，玩家产生影响。

我们如果计算所有光源对所有所有物体的影响，那么这个消耗是巨大的。对于这个例子，我们也是需要用策略解决。我们可以将地图分块，对于我们每个块里面的玩家或者建筑取这个块里面对它们影响最重要的几个光源来计算。通过这样的一个策略，我们就把原来的问题变成了如何选取最重要的光源的问题了。这样的一个问题，又变成一个新的策略问题。我们可以先简单的选用距离来考察一下策略的可用性，来验证我们这个策略选择的正确性。

第三个例子：

当我们客户端有大量玩家的时候，玩家直接会PK，会打怪，会释放很多的技能。这时候我们的客户端又卡起来了，我们又需要使用策略来应对了。当我们再选取策略的时候，我们第一步实际上还是和调试一样需要多观察，只有观察好了，收集到了足够多的信息后，我们才能够想出策略。比如这个场景，当我们发现是特效过多导致客户端卡的时候，而我们又观察出30个特效和50个特效的表现实际上没有什么差别，这时候我们就可以得出一个我们可以使用的策略。我们可以限制同时播放的特效来减少过多的消耗系统资源。

第四个例子：

当我们开服的时候，服务器一时间涌进了大量的玩家。因为玩家的数目太多，导致登录的时候占用了太多的CPU资源，从而影响了主线程的逻辑时间。这个案例和前面的类似，只是发生在服务器端。我们可以用一个队列，将需要登录的玩家放在一个队列中，在指定的时间内进行有序的加载。这样，这块登录的时间就变得可控制了，也不会影响主逻辑的时间。我们经常看到有些游戏登录要排队好几小时，也就可以理解了。

上面4个例子，都是在用策略来解决系统的各种问题。总结下我们得出策略的方法：

先观察问题，看看是否能够直接减少问题的规模解决。

尝试将问题通过策略来简化，缩小到策略的选择上面。

如果简化不成，将问题的影响分担到后续的若干时间内，避免影响的高峰。

5. 报警机制

报警机制常见于后台应用中，这些后台应用当出现异常的时候，需要紧急通知到项目的负责人。在游戏中，如果服务器开不起来，运维的负责人也要紧急通知到负责人的。报警的意义就在于在最快的时间内减少损失，解决问题。曾经有一次服务器进行维护更新，更新后服务器验证完基础的问题后，一直跑的顺畅。但是玩家突然发现重新登录后数据丢失了，一直丢失。因为发生在半夜，这个问题直到隔了10小时才通知到了游戏方。这个问题给我们带来的思考有2点：

如何预防此类问题。

出了问题需要及时通知游戏方。

第一点始终是我们最需要去思考的。我们可以设计机制来规避逻辑，我们可以设计机制控制影响范围，我们可以加强测试环节等等。但是逻辑错误的本源是人并不总是细心的。无论是什么样的程序员，无论测试了多少次，代码终究也可能在极端的情况下藏着某些bug。

第二点是我们这个案例需要重点关注的内容。我们允许产生bug，但是我们希望能在第一时间控制住这个bug的影响。那么我们希望能在一些情况下得到通知。无论是短信通知或者是电话通知，这部分的内容交给运维去解决。那么我们需要做的事情是定义清楚什么情况需要做报警，以及报警的时候应该怎么处理。

我们来看看什么情况需要报警。服务器短时间内多次重启需要报警，这种属于比较大的清晰可见的情况。往内一点讨论，当玩家存储的数据存储失败的时候，不一定需要报警。而当多个玩家出现这个情况时，就需要报警了。逻辑本身可能是会发生错误的，但是这种错误可小可大，我们不需要每个错误都去报警，这样会干扰我们正常的开发。我们尝试再加入触发次数这样的条件来定义一个报警事件。当多个玩家在一段时间内发生了多次值得报警的bug的时候，比如存储数据失败，登录失败，充值失败，在游戏中无法移动等情况，我们都可以进行报警。报警采用的游戏处理手段可以是停止新玩家登录，也可以是关服等。

所以最终我们需要做的就是根据项目情况，制定出一些报警条件。这些条件一般是现有的游戏框架无法规避的且必然会出现的问题。

6. 游戏更新

游戏中的更新包含客户端的更新和服务器的更新。对于服务器的更新，我们可以安排停服维护，将修改好的代码或者配置替换到服务器上。我们还可以使用热更新的方式，不停服对出错的代码或者错误配置进行更新替换。对于客户端的更新，我们通常是更新对应的资源和脚本。客户端的这些更新一般是等下次启动客户端的时候再生效。

下面我们讨论几个东西：

更新原则是什么？

服务端与客户端需要同时生效怎么处理？

上百个服务器如何更新？

第一个是更新原则，我们通过一个案例来分析。我们需要修复服务器的一个bug，需要更新代码以及数据库。首先我们要意识到更新代码和更新数据库这2个操作不一定是可以同时生效的操作。当我们同时去执行他们的更新操作的时候，有可能代码先生效了，也有可能数据库先生效了。如果是代码先生效了，那么代码可能还会调用更新需要的数据库过程，这时候数据库过程实际上没有生效，那么就会引发错误。从这个角度看，我们进行更新的时候应该是先更新基础层，再更新上面的逻辑层。还有一种可能，比如说数据库的数据字段变更了。这时候即使是先更新基础层，依然会存在代码与数据对不上的情况。在商业环境下，我们一般情况如果是遇到需要修改字段的，那么我们会保留旧的字段，新增一个字段，用新增来替代变更。

第二个是服务端的更新与客户端的更新如何同时生效。需要同时生效是因为服务端和客户端都可能更新配置，而配置我们希望是两端读到的都是一样的。服务端的更新我们可以做到瞬间就能生效，但是客户端的代码资源都是放在CDN上，更新生效的时间是不一定的。那么我们可以看到的是同时生效的难题在于客户端资源的生效时间是不确定的。我们需要引入资源的版本机制，就像前面说的缓存更新一样。我们在CDN生效之后，再更新服务端，继而切换资源的版本号，通过这样来达到它们的同时生效。

第三个是当我们服务器的数量变多的时候，我们希望能够对大量的服务器同时做更新。这个的支持我们应该放在后台支持模块。通过运维对多个服务器进行通知来达到批量更新的目的。

事实上，更新，热更新不止上面看到的这些情况。比如对于我们客户端，当我们修改完窗口脚本，我们希望重新打开就能立即生效，这样我们的开发效率就大大提高了。所以，当发生变更情况的时候就意味着更新。更新不仅仅是服务端客户端的更新，更是整个工作流程的更新。

7. 硬钢，使用硬能力解决问题

这一节我们讲的是如何解决那些难缠的问题。有些问题我们通过策略都无法解决的。它常常是外部原因，或者出现比较困难的排查。

我们在使用C#的时候，有些函数我们是一定会产生gc的。当我们追求极致的性能的时候，gc就成为了一个新的绊脚石。如果是一个频繁调用的函数，当它一直产生gc的时候，这个问题就必须要解决了。这种问题实际上已经超出了逻辑的范畴，我们需要用硬实力解决。对于gc问题，我们只有通过更底层的语言来规避。我们可以对需要gc的接口使用C++实现，这样才能应对这个问题。

我们在打开某个功能的时候发现系统出现了卡顿。经过很多的排查，依然找不到问题所在。这个时候我们就只能使用注释大法了。将代码进行二分注释，一步一步的排查问题。

我们在游戏上显示的一个模型，在大部分机子上面显示正常，在特殊的一些机型上显示异常。因为涉及到渲染环节，没有成熟的调试方法。那么我们就采用最原始的方法。在shader的代码中每一行输出下当前颜色，与正常的渲染对比，排查不兼容的原因。

还有一些引擎内的gc，比如unity的setActive这样对对象进行显示隐藏的接口。这种接口我们非常常用，我们也无法用C++去实现了。这种情况我们只能通用别的方式去实现。比如说我们不隐藏对象了，而是移到到屏幕之外。有些操作非常耗费CPU，产生的卡顿无法接受，而它又是必须调用的api。这种情况我们可以开线程，把卡顿的部分放到线程里面执行。唯一要处理好的就是线程之间的交互。

第七章

提高能力的方法

1. 初步

我们在前面的章节中，主要强调了对于当前工作中重复的以及变化频繁的部分做各种各样的抽象与总结。在有效的总结后得出可以跨项目或者可轻易复制的技巧、模式和框架。这些行为大部分是发生在工作之中，通过重复性的代码编写产生的。那么这一章我们将描述一下如何在这个过程中再进一步提高自己的能力。我们将以白鹭引擎为例，来讲解一下这个过程是如何提高自己能力的。

第一步我们需要下载好白鹭的基本开发环境，并把它的周边产品玩起来。比如它的可视化编程工具、粒子编辑器等。了解了它是什么，大概有什么后，我们就可以来了解下如何使用它来开发。

我们先应该新建一个工程，跑跑它的demo，看看它运行的入口，以及资源，代码等的管理方式。这个阶段对于一个项目的主程，他需要去看看性能等问题。如果是刚入门的开发者，只需要好好的熟悉它的使用即可。最后我们要把这个工程运行起来，并做一点个性化的修改，来完成实际对它的修改。

经过了前面2步，我们就进行项目组正式开工了。这时候我们会不停的写大量的代码。这时候对于新人而言，即使开发了2-5个项目，也还没到阅读源码的时候。这个时候需要配合阅读文档来提高。通过阅读官方文档来了解引擎本身的设计结构，并建立印象。

经过了前面几个项目的历练，后面的项目对我们而言大部分就是类似的逻辑。这时候我们可以开始通过阅读源码提高自己能力了。对于阅读代码这件事，和入手一个引擎的使用是类似的。首先就是要把它玩起来。通过了解它的使用，API相关的说明来了解它的外在。再通过一些问题打入它的内部机理。常见的有个错误的阅读方式有

1. 下载代码后立刻直奔源码进行阅读。

2. 在未了解如何使用的情况下直接百度阅读相关的源码分析。

这些阅读方式看似直接是深入到代码内部，实际上却得不到相应的提高。那么我们建议的阅读方式是先从使用入手，再到了解内部机理。就像前面介绍的几个步骤一样，讲究的是顺理成章。

准备开始阅读代码之前，我们先要做的是整理问题。这些问题是你之前开发过程中遇到的或者绕过的地方，或者是感兴趣的地方。我们先简单整理一下：

1. 引擎底层如何对同个贴图进行合并批次的操作？（兴趣）

2. 引擎底层是如何渲染了？（兴趣）

3. 为什么加载资源经常卡住，可能有哪些原因？（疑问）

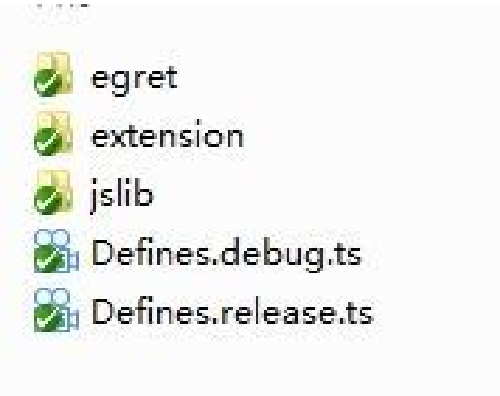
4.label、textField用哪个比较好？（之前不是很在意）

5. tween在低端手机上出现卡顿，是不是存在性能问题？（性能）

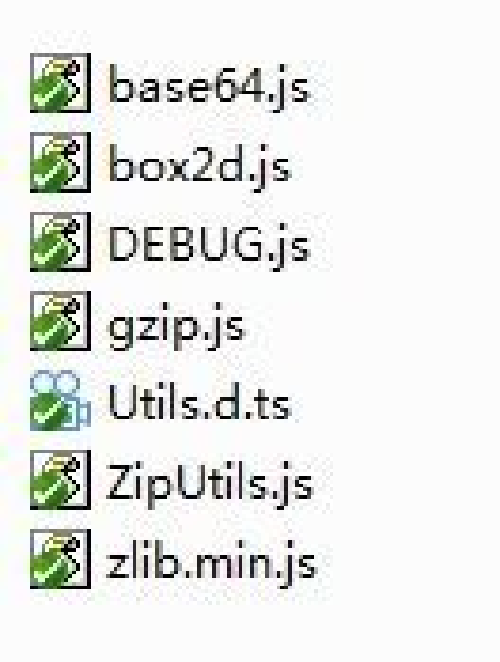
6.有哪些值得学习的模式？（兴趣）

7.如何知道运行时的内存占用情况？（性能）

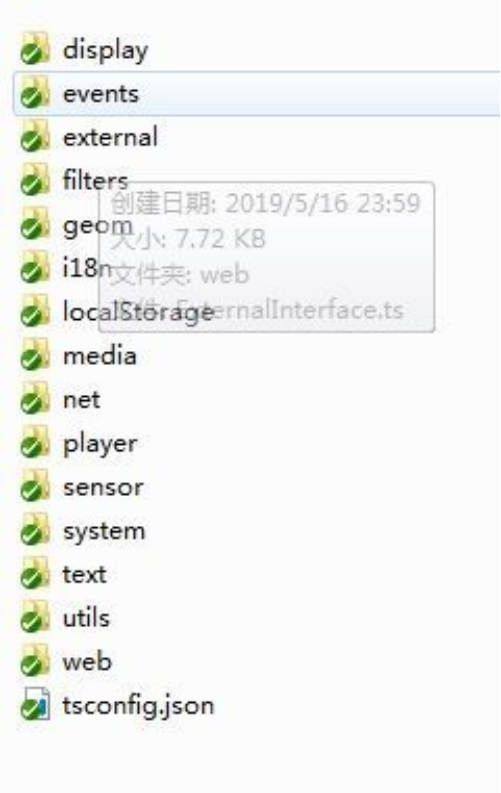
我们可以整理出很多的问题，后面我们将带着这些疑问来了解引擎源码。我们下载好引擎源码（版本5.2.19）后，src目录下是这样的：

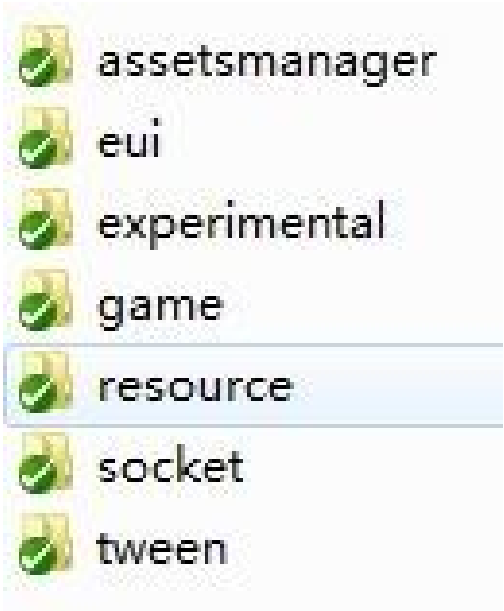


下面的jslib文件夹内容：


base64.js
box2d.js
DEBUG.js
gzip.js
Utils.d.ts
ZipUtils.js
zlib.min.js

egret文件夹内容：





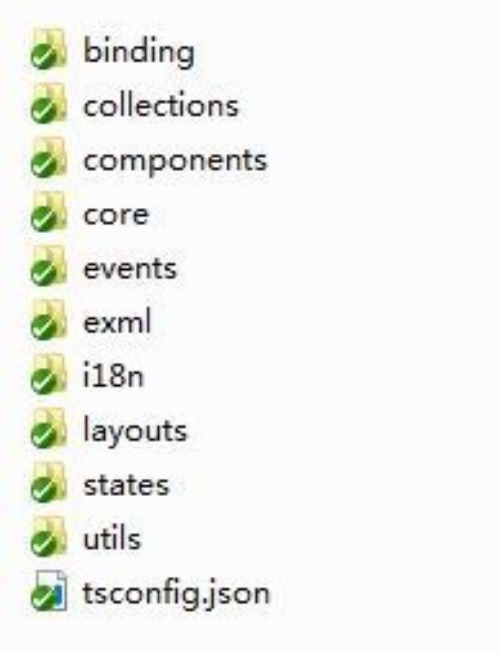
extension文件夹内容：

下一节我们将正式开始阅读里面的代码，并了解其中奥秘。

2. 阅读源码

我们看了代码文件夹中的东西，也列了我们的一些疑问。我们先挑选疑问中最接近我们实际代码的问题，通过这个问题找对应的文件夹。我们说问题：label、textField用哪个比较好这个问题比引擎底层是如何渲染了这个问题更接近我们的工作需求，所以我们先看这部分。这边我们其实需要对问题进行一个排序，可以按照和工作内容相关来排序，也可以按照其他排序方式。因为我们推荐的方式是从外到里，所以我们按照和日常工作的相关性来排序。

label是属于eui这个组件的，因为我们在使用的时候经常会在前面加上它的命名空间eui。在前面的文件夹目录中，我们看到了它属于extension下。extension顾名思义，就是拓展的，不一定需要使用的东西。也就意味着在白鹭引擎提供的开发框架下，你是可以不使用这个eui库来开发的。我们继续看看这个库里面的情况（如下图）：



看到eui库的情况，我们先要根据我们的经验去猜测一下这些内容。最显眼的是tsconfig.json文件，它是这里面现在看到的唯一一个文件而不是目录。我们看到config相关的字眼，我们可以猜想它是某个配置问题。配置文件往往关联着整个目录的结构以及对应的编译工具。接着我们从上到下阅读整个文件夹的情况：

bingding，直接翻译就是绑定。猜想可能是数据绑定相关。

Collections，集合。这个地方有意思的是typescript已经提供了数组和字典，我们平常也没有什么额外的集合需求，那这个集合有什么作用呢？这个问题我们可以加入到问题列表，等后面再看。

components，组件。这里很可能就是我们要找label的地方，因为我们的label在编辑器里面也是放在组件下面的。

core，核心。猜想是这个文件夹里面用到的一些最基础的方法或对象。

events，事件。eui用到的事件都应该在这里了。

exml，某种xml的格式。因为我们平常在制作皮肤的时候，生成的文件的后缀就是exml，所以我们可以猜想它是皮肤文件的解析支持文件。

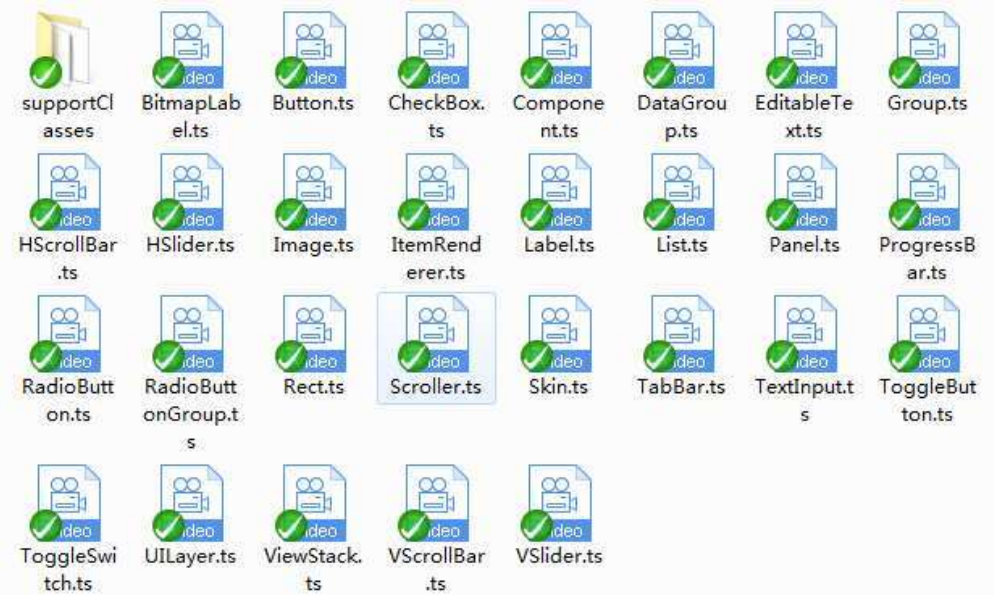
i18n，不确定的东西。回头再看。

layouts，布局。在编辑器中我们经常会用到这个东西。

states，状态。也是要放后面看，暂时只能猜测是状态相关，也行是按钮的各个状态之类的？

utils，工具类。

从上面的文件阅读过程中，我们大概猜到了label的位置，也新生成了一个问题。接



着我们就进入component探个究竟。Component文件夹里面的内容如下：

我们直接打开label，来阅读label的代码。阅读代码的时候，要留意它的注释。比如在label的开头，我们可以看到这样的注释：

* Label 是可以呈示一行或多行统一格式文本的UI组件。要显示的文本由text 属性确定。文本格式由样式属性指定，例如 fontFamily 和 size。

* 因为 Label 运行速度快且占用内存少，所以它特别适合用于显示多个小型非交互式文本的情况，例如，项呈示器和 Button 外观中的标签。

* 在 Label 中，将以下三个字符序列识别为显式换行符：CR（“\r”）、LF（“\n”）和 CR+LF（“\r\n”）。

* 如果没有为 Label 指定宽度，则由这些显式换行符确定的最长行确定

Label 的宽度。

* 如果指定了宽度，则指定文本将在组件边界的右边缘换行，如果文本扩展到低于组件底部，则将被剪切。

注释是辅助我们阅读的利器，越是好的代码，注释也越多。我们接着阅读:export class Label extends egret.TextField implements UIComponent,IDisplayText {

上面这行告诉我们Label继承自TextField且实现了UIComponent，IDisplayText这2个接口。知道了Label是继承自egret.TextField，那么它们的区别就是Label剩下的其他代码了。这边又延伸出了一个问题，我们需不需要去看一下UIComponent,IDisplayText这2个接口里面有什么。实际是不需要的，因为它们的内容和我们的问题无关，我们要紧紧抓着我们的问题作为我们理解的方向。我们接着看后面的代码，后面的代码直接出现的是style相关的，核心代码：

```
$setStyle(value: string) {

    if (this.$style == value) {

        return;

    }

    this.$style = value;

    let theme: Theme = egret.getImplementation("eui.Theme");

    if (theme) {

        this.$changeFromStyle = true;

        for (let key in this.$revertStyle) {

            this[key] = this.$revertStyle[key];
```

```
            }
            this.$revertStyle = {};
            if (value == null) {
                this.$changeFromStyle = false;
                return;
            }
            let styleList = value.split(",");
            for (let i = 0; i < styleList.length; i++) {
                let config = theme.$getStyleConfig(styleList[i]);
                if (config) {
                    for (let key in config) {
                        if (this.$styleSetMap[key]) {
                            let revertValue = this[key];
                            this[key] = config[key];
                            this.$revertStyle[key] = revertValue;
                        }
                    }
                }
            }
            this.$changeFromStyle = false;
        }
    }
```

这个函数里面用到了几个变量：$style，$revertStyle，$changeFromStyle，$styleSetMap我们需要关注这几个变量，方便我们理解整个函数在做些什么。

这时候要上下文看看，也就是把函数的前后看看。我们向前看会看到：

```
public get style(): string {
    return this.$style;
}
public set style(value: string) {
    this.$setStyle(value);
}
```

我们可以得出，$stype暂时没做什么，仅仅提供给外界进行获取与设置。往下看会看到：

```
$setFontFamily(value: string): boolean {
    if (!this.$changeFromStyle) {
        delete this.$revertStyle["fontFanily"];
        this.$styleSetMap["fontFanily"] = false;
    }
    return super.$setFontFamily(value);
}
```

这边用到了$changeFromStyle这个变量，当它的值为false的时候，它就和父类的setFontFamily产生了区别，这也是label与textfiled的一个区别。找到这个区别后，我们就要尝试反推刚刚函数的意图。先慢读一下这一小段：

```
if (!this.$changeFromStyle) {
    delete this.$revertStyle["fontFanily"];
    this.$styleSetMap["fontFanily"] = false;
}
```

意思是：当this.$changeFromStyle为假的时候，删除this.$revertStyle这个字典里面"fontFanily"这个索引的值，并且设置this.$styleSetMap字典["fontFanily"]索引为false。

这边我们可以得出几个信息，this.$revertStyle、this.$styleSetMap都是字典，且2个可能存在关联关系。因为this.$styleSetMap存的值是真与假，相对另外一个字典而言，它的重要性很可能仅仅是用于判断。

看完这个函数，再次回去看$setStyle这个函数，我们可以要继续慢慢的读：

```
let theme: Theme = egret.getImplementation("eui.Theme");
```

读到这里，我们结合下面theme的使用的地方：

```
let config = theme.$getStyleConfig(styleList[i]);
```

我们可以暂定标记这里的theme的作用就是获取某个stype的配置。标记有点类似扫雷里面用到的思维，是一种尝试，指暂定某个东西的意义，延后处理它。我们接着看下面的代码：

```
this.$changeFromStyle = true;

for (let key in this.$revertStyle) {

    this[key] = this.$revertStyle[key];

}

this.$revertStyle = {};

if (value == null) {

    this.$changeFromStyle = false;

    return;

}
```

我们看到$revertStyle在被使用后就可以赋值为{}，这个表明了它很可能是一个还原操作。后面对value的判断比较简单，就是没有设置的情况下

$changeFromStyle 就不生效。这一小段暂时读到这里，继续读下面的代码：

```
let styleList = value.split(",");

for (let i = 0; i < styleList.length; i++) {
```

这一段注意这里的value是传入的参数，它被分割后是个数组，可以猜测它应该表示的是多个style。后面是下面这段代码。基于前面的假设，这里就取到了要设置的style的配置。

```
let config = theme.$getStyleConfig(styleList[i]);
```

如果配置存在的话：

```
for (let key in config) {
    if (this.$styleSetMap[key]) {
        let revertValue = this[key];
        this[key] = config[key];
        this.$revertStyle[key] = revertValue;
    }
}
```

所有在这个配置里面的key，如果$styleSetMap里面有这个值，那么进行里面的操作。里面保留了原来this上的key的值（保存到$revertStyle中），并赋值给一个新值。我们刚刚说的上下文环境，在下面我们发现了：

```
$setFontFamily(value:string): boolean {
    if (!this.$changeFromStyle) {
        delete this.$revertStyle["fontFanily"];
        this.$styleSetMap["fontFanily"] = false;
    }
    return super.$setFontFamily(value);
}
```

往上我们会看到：

```
/*** style中属性是否允许被赋值，当主动赋值过属性之后将不允许被赋值*/
private $styleSetMap = {
```

```
        "fontFamily": true,
        "size": true,
        "bold": true,
        "italic": true,
        "textAlign": true,
        "verticalAlign": true,
        "lineSpacing": true,
        "textColor": true,
        "wordWrap": true,
        "displayAsPassword": true,
        "strokeColor": true,
        "stroke": true,
        "maxChars": true,
        "multiline": true,
        "border": true,
        "borderColor": true,
        "background": true,
        "backgroundColor": true
    };
```

引擎正好也存在注释说明，$styleSetMap的作用就是使得主动赋值过属性之后将不允许被赋值。

整理下上面说的所有的思路，大概可以理清楚这段函数干的事情：

1. 重复设置style会把旧的style先还原，再设置新的。
2. $styleSetMap的作用就是使得主动赋值过属性之后将不允许被赋值。

3. 设置style仅仅是修改了this上面的一些key的值，并没有做其他操作。

前面得出的是现象，进一步需要尝试思考。如果一个set操作并没有做一些你认为该做的行为，仅仅是改变一些值的话。那么说明有其他操作（通常是每帧的要做的事情）会来取它的值，并且做出表现（比如说渲染出来）。

再接着的是 $setFontFamily函数的不舒服的实现。因为setStyle会导致一些内存放在了$revertStyle中没有销毁，所以其他一些函数重载了父类接口替他清理这块没有办法及时释放的内存。

通过上面这部分的解读，我们这部分的label与textfiled的区别紧紧是为了有机会清理一些内存。当然，label和textfiled函数存在其他差异的。接下来我们不一部分一部分看了，而是想想哪个函数是最经常用的。setText是最经常用的，所以我们直接看它做了什么。它的实现如下：

```
$setText(value:string):boolean {

        let result:boolean = super.$setText(value);

PropertyEvent.dispatchPropertyEvent(this,PropertyEvent.PROPERTY_CHANGE,"text");

        return result;

    }
```

这段比较好理解了，唯一的不同就是子类对分派了事件。我们可以应用上面的思维在后面的代码中继续比对了。我们再比对一个重要的也是常有的：

```
$setWidth(value:number):boolean {

    let result1:boolean = super.$setWidth(value);

    let result2:boolean = UIImpl.prototype.$setWidth.call(this, value);

    return result1 && result2;

}
```

差异在于多了UIImpl的$setWidth的操作。这里可以说明lable和textfield的差别里面肯定有一个UIImpl类。我们跳转到UIImpl的实现，可以发现它的关系链为：

```
    export class UIComponentImpl extends egret.DisplayObject implements eui.UI
Component
```

DisplayObject具备布局等功能，所以布局也是lable和textfiled的差异。我们继续往下读就会发现它们所有的不同了。我们停在这个地方，对这个小节进行总结：

1. 通过对比法可以找到2个父子类之间的差异

2. 差异化的函数可以通过上下文（这边主要指同个文件的上下几行）

来反推具体的实现。

3. 读代码可以慢慢细读的，需要用到合适的快捷键来辅助定位。

4. 可以通过常用的函数，快速定位到父子类的不同点。这里常用的指的是工作相关的，经常遇到的。

经常在一些QQ群或者论坛里面，很多老手会劝新手不用学那么多框架，而是用什么学什么。这个思想也是需要应用到阅读源码之中的。日常常用的东西大部分是枯燥的，但是它却可能是最容易抵达核心的有利工具。

最后再一次按照正向思维总结：

1. 寻找最熟悉的点作为入口。

2. 卡住时寻找上下文，不断挖掘其他信息。

3. 放慢速度，精准阅读。

上面的几个步骤与我们前面的章节中的调试用的思维方式都是一样的，这也正是很多主程序可以通吃很多内容的原因，本质都是因为这里面用到的思维都是差不多的。

3. 深入理解差异化

我们前面在阅读文件目录的时候，发现一个文件夹是collections。这个文件夹在之前是一个疑惑，因为我们现有的数据结构已经可以满足我们的需求了，但是引擎却提供了额外的一个容器，它是ArrayCollection类。我们看一下它的注释：

ArrayCollection 类是数组的集合类数据结构包装器，可使用

<code>ICollection</code>接口的方法和属性对其进行访问和处理。

使用这种数据结构包装普通数组，能在数据源发生改变的时候主动通知视图刷新变更数据项。

我们从上面的注释中已经可以得知它的用途，也就是它需要去主动通知窗口，通过派发事件的形式。结合前面我们讲的label与textfiled的一个差异是事件。我们可以得出，事件在白鹭引擎中起到非常重要的作用。再结合我们第二章对于事件通知的理解，我们就知道了白鹭引擎底部实际上是用事件的方式来做到解除对象间的耦合的，并给对象提供了通知以及注册通知的机制。

对于这个例子，我们需要详细的区分哪些是逻辑层提供的机制，哪些是语言层提供的机制。对于typescript中的字典，它本身是不提供类似于数据变更这样的机制的。我们很难得知一个字典里面的数据发生了变化，同理我们也很难得知一个数组里面的数据项发生了变化。对于lua语言而言，它本身提供了元表等机制支持了对于数据变更的观察。综合大量的计算机语言的设计，我们知道数据变更并不是一个通用的语言层机制。因为底层语言的非通用性，所以我们应该把这一套机制建立在逻辑层。在这边的例子中，白鹭引擎建立了这套底层机制提供给开发者使用，我们在它的上层进行代码书写时，也应该使用统一的通知机制，使得逻辑层代码可以尽可能的提供一致性的预期。

细读ArrayCollection类，我们看到它额外派发了：

CollectionEventKind.ADD

CollectionEventKind.REMOVE

CollectionEventKind.REPLACE

CollectionEventKind.UPDATE

4个事件，对于这些事件的派发选择可以供我们学习一下。这些是从窗口组件的需求中抽离出来的。从底层看代码设计我们可以反推出一些需求。而如果我们从上面的使用层面看，我们也可以尝试总结出底层设计的规律。这是一个双向的对应过程。

我们聊的差异化，通常是因为下面几个原因产生的：

1. 性能优化导致的。常见的发生于很多开源代码中，它们会自己实现一套对应的数据结构，如字典、数组、栈等。通过新写的数据结构来替换引擎或者语言层提供的数据结构来加快执行效率。

2. 性能分析导致的。常见的发生在项目的网络、异常、函数执行等模块。通常用于对整个项目的内存情况，函数的执行时间进行分析。

3. 封装导致的。常见的是将一些接口统一用类的形式组织，并封装成跟项目风格一致的代码。封装一般也会提供一些校验，用于提前报告错误，防止代码问题的扩散。

4. 抽象导致的。常见的是将一些接口进行统一化处理，对上面层的调用提供统一的接口访问形式。

5. 机制导致的。为了提供更方面，更不容易犯错的代码框架，通过不断的进行封装以及抽象，并设计相应的代码实现规范，来达到开发效率与性能的最大化。

我们以前面讨论过的ArrayCollection来看看它对应的差异化内容。它提供了一些事件进行派发，这些事件是白鹭引擎内的组件互相通信的基础。所以它主要是因为机制导致的。当然，它还存在的其他一些差异化：

```
    public addItemAt(item:any, index:number):void {
        if (index < 0 || index > this._source.length) {
            DEBUG && egret.$error(1007);
        }
        this._source.splice(index, 0, item);
        this.dispatchCoEvent(CollectionEventKind.ADD, index, -1, [item]);
    }
```

可以看到它对传入的参数进行了校验。这也是常见的一种封装差异。

4. 阅读接口设计

我们前面聊到了差异化，聊到了抽象。抽象产生的差异化在代码中的表现通常表现在接口的实现上。对应我们前面在框架设计中的内容就是”子类实现的部分”。

我们在刚开始接触别人代码或者一个引擎的时候，我们关注的很重要的一点就是父类提供了哪些机制（或者说可覆盖的方法）给子类去拓展或者说实现差异化。子类拓展指的是像框架章节说的提供给框架去实现的方法。差异化指的是提供给更多的不同的子类实现自己个性化需求。我们可以将父类提供的接口用作不同的用途，这取决于我们的需求。

我们来看一下白鹭引擎里面比较重要的一个类的设计，位于extension\eui\core下的UIComponent。UIComponent 类是所有可视组件（可定制皮肤和不可定制皮肤）的基类。这个类继承自egret.DisplayObject。这个地方我们首先要留意一下，它们所属的命名空间已经不同了。UIComponent位于eui下，DisplayObject位于egret下。这种命名空间的不同，可以看出eui在拓展egret方面一定提供了更多不同的机制。

接着我们来看看UIComponent下的接口：

```
// * 创建子项，子类覆盖此方法以完成组件子项的初始化操作，

// * 请务必调用super.createChildren()以完成父类组件的初始化

// */

// protected createChildren():void{}
```

这是第一个接口，它强调了子类必须调用super.createChildren()以完成父类组件的初始化。这个注释说明了这个接口是为了拓展机制而生的接口，只有为了拓展机制我们才一定要调用父类的同名接口。同时，在UIComponent下，一定还存在的其他子类提供了额外的机制。

我们再看看这个类提供的其他子类应该关注的接口：

```
protected commitProperties():void{}

protected updateDisplayList(Width:number,Height:number):void{}

protected invalidateParentLayout():void{}
```

这几个接口它们的注释没有说需要调用父类的同名接口，说明它们的目的只想做差异化。这个差异化在它的下一个子类就应该得以区分。接着我们来谈谈这2种接口在设计上面的一些关键点。首先，我们看到一个是拓展框架，一个是拓展子类的差异化。拓展框架的接口为的是更好的服务于机制本身，它更强调的是满足更多的子类，子孙类进行拓展。而子类差异化的实现，更多服务于项目的快速开发，减少重复的代码。

我们再来看一下UIComponent的一个直接子类UIComponentImpl的实现，它的注释是这样的：它是EUI 显示对象基类模板。仅作为 UIComponent 的默认实现，为egret.sys.implemenetUIComponenet()方法提供代码模板。在此类里不允许直接使用super关键字访问父类方法。一律使用this.$super属性访问。

它里面有2个接口我们需要关注下：

```
/**

* @private

* 子类覆盖此方法可以执行一些初始化子项操作。此方法仅在组件第一次添加到舞台时回调一次。

* 请务必调用super.createChildren()以完成父类组件的初始化

*/

protected createChildren(): void {}

/**

* @private

* 子项创建完成，此方法在createChildren()之后执行。

*/

protected childrenCreated(): void {}
```

这个地方很有意思了，有两个名称接近的函数。很多人经常看代码会出现这种疑惑，不明白为什么会有雷同的接口。我们看下它们的注释，其中一个需要调用父类接口，一个不需要。所以一个是拓展框架用的，一个是给子类直接实现用的。从设计层面的角度而言，这个类在这里的两个不同接口就是为了划清楚框架拓展和子类差异化之间的界限。它是可以用单独一个接口createChildren来做到相同的事情，只是这样会产生混乱。额外提供的childrenCreated提供给子类差异化一个不需要考虑是否需要调用父类同名接口的机会，大大的减轻了逻辑对象的使用负担。

在底层代码中展现出来的这种对接口的设计告诉我们，我们需要区分框架层和逻辑层。更大层面上面告诉我们的是我们应该理清楚层与层之间的关系。只有界定了它们的关系，将它们更好的独立来实现，并减少它们之间的使用依赖，才能使得高效的开发得以实现。

5. 后续

前面的一些小结我们理解了一些框架上的设计，对照着我们日常的逻辑进行了梳理。后面剩下的内容就是细节了。比如我们看下UIComponent的一个直接继承类BitmapLabel对它的拓展。BitmapLabel之所以没有直接继承自上面提到过的UIComponentImpl是因为UIComponentImpl是代码作者提供的一个参考，并不是强制的父类实现。这种参考通常说明了在这个地方暂时没有办法找到一个最佳的实现来完全囊括后续的子类需求，所以只能提供一个参考给后续子类借鉴。

在BitmapLabel的createChildren中：

```
protected createChildren():void {

    if (this.$fontChanged) {

        this.$parseFont();

    }

    this.$createChildrenCalled = true;

}
```

我们看到它在子对象创建好的时候尝试解析字体，并标记了一个状态。我们前面提到这个createChildren可能是异步的。所以对于一个异步的过程，这个过程中可能发生了很多情况，比如这个文本类可能字体发生了改变，它都需要用到这个时机来修正自己的状态。

同理，我们可能在其他类中也去分析它们自身的细节实现。我们这本书主要希望告诉大家的是如何去思考，并不想过多的讨论细节。但是，这并不意味着细节不重要。当深入到一个难题最底端的时候，就是积累细节的时候。同时为了更好的理解上层设计，我们也需要细节。细节与整体抽象是个互补的过程。

理解细节的难点在于细节无止境，它的末端可能是个二进制问题，也可能是个数学问题。一直往下探索是无止尽的，但这个过程中所蕴含的抽象思想都应该被尝试理解。这些思想和逻辑部分用到的思想很多都是相同的，大部分都是性能和效率之间的比拼。希望读者根据自己的疑问和兴趣再进行适合自己的阅读。

结束

在一个大型项目中，加班熬夜通宵都是常事。在游戏功能模块滚滚向前开发的过程中，有的项目上了被砍掉，有的上了退回来重来，有的成功上线。退回来的项目再调整再调整再调整，有机会的话就一直上去测试。

整个的项目过程，项目的开发周期是非常紧张的。项目紧张，就有着各种各样强压的需求。需求多了，加班多了，抱怨是难免的。也许可以以项目未来为鼓励，也许可以申请加薪，但是当一个不合理的状态持续的存在较久时，就会有人质疑，有人离开。离开、进入使得人员的迭代速度发生了变化。为了留住更多的人，我们会建立友谊来凝聚整个团队。但是当外界压力越发爆炸时，内聚的东西也开始散了出去，维护一个整体的方式变成了快速培养新人来进行迭代。

在程序里面，我们希望一个一个组件之间低耦合，最好的方式就像插件一样，需要的时候装上，不需要的时候卸下。而在管理方面，同样的管理逻辑也许可以存在在一些大型公司里面，他们希望人员可以即插即用。但是在一个项目团队里面，耦合也许是种更好的团队形态。构建一个好的游戏以及赚钱是一个团队出发的初衷，但是这个过程没有人关心，大家只关心结果。低耦合的团队方式并不影响这个初衷，但是人本身也有初衷，当这两个初衷发生了冲突的时候，那就是一个新的开端。

项目上线的人，每周一个版本。熬着的人在等着爆发，退的人在休养生息。我想只有尝试不断变好才能在这个流变中稳如泰山。本书无法解决高压下的项目管理问题，这也是本书生产的原因之一。它只能告诉你尽量在这个流变中控制好时间，把握好力度。