**QFramework系统设计架构分为四层及其规则：**

- 表现层：ViewController层。IController接口，负责接收输入和状态变化时的表现，一般情况下，MonoBehaviour 均为表现层
  - 可以获取System
  - 可以获取Model
  - 可以发送Command
  - 可以监听Event
    - 不能发送Event
- 系统层：System层。ISystem接口，帮助IController承担一部分逻辑，在多个表现层共享的逻辑，比如计时系统、商城系统、成就系统等
  - 可以获取System
  - 可以获取Model
  - 可以监听Event
  - 可以发送Event
    - 不能操控command
- 数据层：Model层。IModel接口，负责数据的定义、数据的增删查改方法的提供
  - 可以获取Utility
  - 可以发送Event
- 工具层：Utility层。IUtility接口，负责提供基础设施，比如存储方法、序列化方法、网络连接方法、蓝牙方法、SDK、框架继承等。啥都可以干，可以集成第三方库，或者封装API
- 除了四个层级，还有一个核心概念—Command
  - 可以获取System
  - 可以获取Model
  - 可以发送Event
  - 可以发送Command
- 层级规则：
  - IController 更改 ISystem、IModel 的状态必须用Command
  - ISystem、IModel状态发生变更后通知IController必须用事件或BindableProperty
  - IController可以获取ISystem、IModel对象来进行数据查询，可以操控command
  - 上层可以直接获取下层，下层不能获取上层对象
  - 下层向上层通信用事件
  - 上层向下层通信用方法调用（只是做查询，状态变更用Command），IController的交互逻辑为特别情况，只能用Command

- 上面说的event是全局的event，不属于任何脚本对象，SendEvent的行为会触发所有订阅了该Event的函数

| | IBelongToArchitecture | ICanSetArchitecture | ICanGetModel | ICanGetSystem | ICanGetUtility | ICanRegisterEvent | ICanSendCommand | ICanSendEvent | ICanSendQuery |
|---|---|---|---|---|---|---|---|---|---|
| System | √ | √ | √ | √ | √ | √ | | √ | |
| Controller | √ | | √ | √ | √ | √ | √ | | √ |
| Command | √ | √ | √ | √ | √ | | √ | √ | √ |
| Model | √ | √ | | | √ | | | √ | |

- System通过监听/发送事件影响游戏进程，但它不发送command，让controller分担这部分逻辑
- model属于底层对象，使用BindableProperty可以让获得model的对象监听model的数据变化，所以它可以发送数据
- command也是底层对象，但它是执行者，所以几乎是全能的，但因为他是一次性的，不是存在性的，不能监听事件
- controller不能被注册，只能设定其所属的Architecture，它通过发送command影响游戏数据和状态，它可以监听事件，但不能发送事件；Controller一般是MonoBehavior挂载到Go上
- Controller是MB，挂载到GO上外界可以看到的，所以Controller是与外输入的交接口，而System\Model\Command都是内部的状态
- 其他
  - Query类似于vue里的Getter，可以计算出各个model的组合数据，主要提供给查询用，
  - Utility不继承任何接口，做一些辅助的行为比如打印日志，存储数据等

## 参考链接

- 【OpenSourceC#】Unity框架-QFramework_PartnerLv的博客-CSDN博客

## 个人评价

- 比较适合小游戏
- 对于使用反射实现的IoC容器，暂时持保留意见，因为反射据说比较耗性能
  - 个人认为可以改进，添加一个接口，IoCContainner可以从该接口中获取需要的key，将Get时通过typeof获取改为从接口中获取
  - 但可能有点多余，typeof的性能消耗非常小。不使用反射需要改变编程习惯，影响很大，而且使用反射的次数是有限的，并不会每帧都调用Get
- 很好的管理了数据的创建和获取，保证只存在一份数据
- 一个接口定义规则，一个抽象类用于继承
- 有些时候没有必要一定按照该逻辑设计，比如一个模块内部，当模块间有交互时可以用上面的方法指导编程
- 注意model是需要共享的数据，如果数据只在模块内部处理，不使用
- 数据不一定绑定到model，也可能绑定到system
- 最有启发的地方
  - 将共享的数据抽象为Model层，并且为其添加委托，可以像web开发一样监听数据
  - 定义全局的事件，可以灵活地监听事件是否发生
  - command可以是无状态的，只需要声明泛型类型，也可以自己设计构造函数然后传入构造的command对象
  - 通过扩展方法将接口的方法实现统一到一处进行，比如通过扩展类实现了接口ICanSendCommand的SendCommand函数，通过Architecture然后调用Architexture的SendCommand函数
- System主要是分担逻辑，在多个表现层共享的逻辑
  - 编写思路：先按正常方式写controller，然后将controller中对其他模块依赖的部分抽离形成System
  - 将system中的方法和数据暴露在接口中，GetSystem获取的是接口
  - 感觉就是在controller之外抽象出了System层，System层不需要写成MonoBehaviour

## 更多的理解

- 在了解了SO架构、Singleton和static的关系，依赖注入相关知识后再来看QFramework
  - 架构层就是一个泛型单例类，一个游戏可以用很多架构，而且像单例的一种写法一样，在第一次获取时会实例化，并注册（实例化）需要的系统和数据，架构可以从外部获取，像单例一样使用
  - 如果不在架构类的init阶段注册也没有关系，后续的注册流程是一样的，将系统/数据实例注入到IoC容器中，然后调用对应实例的初始化函数
  - 需要注册的只有System（提供可具备状态的方法）/Model（提供数据）/Utility（提供不持有状态的方法），如果不注册，一般就只能通过静态方式获取，而如今使用反射方式动态获取
  - 相当于将Singleton的职责分散出去，不在需要在Singleton内部实现这些方法/数据，通过一个容器让大家自由访问，好处是解耦了，坏处是效率低了，可能原本固定某些单例类能做的事现在将它分散了。但是通过接口。
  - 通过分层实现依赖倒置原则(高层不依赖低层，两者都依赖抽象，也就是接口)，防止违背单一职责原则。这里的分层体现在Model不能获取System，utility谁都不能获取。
    - 层级:System > Model > Utility，所以符合好莱坞原则（我来找你，你别来找我），Model（下层）向System（上层）通信用事件
  - 内存上没有节约，本来在Singleton（此处特指某一类具体的Architexture）内部的数据移到了Container内部管理，Singleton通过接口规范了大家的使用，本来可能可以直接获取到整个model对象，现在获取的都是接口，只能获取属性或方法
  - Controller属于对外的层面，可以访问内部的System/Model/Utility，通过Command改变内部状态，controller可以获取架构层，目的是进入架构层通过接口获取依赖
  - IController其实就是定义了依赖的架构单例，可以获取系统中的数据和方法服务
- QFramework的优秀之处
  - 提供了分层的方案（面向接口编程）
  - 提供了单例解耦的方案，将数据和方法分到不同的类中（通过接口识别所属的架构，注册最重要的就是指定所属架构，好莱坞原则的体现，我来找你），防止违背单一职责原则
- QFramework的不令我满意的地方

    - 使用IoC容器基于反射，性能差
    - IoC容器目的本来应该是为了同一个接口动态绑定不同的实例实现软绑定，但使用中却发现几乎用不上
    - IoC容器也可以不改，只要多做缓存就可以（对于command，可以在构造时传入缓存的model等，因为既然command中能获取model，说明命令发起者也可以缓存model）
    - 基于类型的事件系统使用反射效率较低
- 改进QFramework的思路
    - 将架构层的IoC容器去除不可行，那么只能用多做缓存减少IoC容器带来的负面影响（但要小心实例化的对象变了，一般不会发生，弱引用可能会用到）
    - 缓存Type，Get时传入Type参数
    - 改变分层，更精细化（controller既然能发送command，command能发送事件，那为什么controller不能发送事件）
    - 增加基于枚举的事件系统，弥补基于类型的事件系统的反射效率低问题