

27 | 实战（二）：怎么设计一个“画图”程序？

2019-07-23 许式伟

许式伟的架构课

[进入课程 >](#)



讲述：姚迪迈

时长 10:28 大小 9.59M



你好，我是七牛云许式伟。

上一讲开始，我们进入了实战模式。从目前看到的反馈看，我的预期目标并没有达到。

我复盘了一下，虽然这个程序看起来比较简单，但是实际上仍然有很多需要交代而没有交代清楚的东西。

我个人对这个例子的期望是比较高的。因为我认为“画图”程序非常适合作为架构实战的第一课。“画图”程序需求的可伸缩性非常大，完完全全是一个迷你小 Office 程序，很适合由浅及深去谈架构的演进。

所以我今天微调了一下计划，把服务端对接往后延后一讲，增加一篇 “实战（中）” 篇。这个 “中” 篇一方面把前面 “实战（上）” 篇没有交代清楚的补一下，另一方面对 “画图” 程序做一次需求的迭代。

MVP 版画图程序

先回到 “实战（上）” 篇。这个版本对画图程序来说，基本上是一个 MVP 版本：只能增加新图形，没法删除，也没法修改。

怎么做？我们先看 Model 层，它的代码就是一个 [dom.js](#) 文件。从数据结构来说，它是一棵以 QPaintDoc 为根的 DOM 树。这个 DOM 树只有三级：Document -> Shape -> LineStyle。具体细节可以参阅下表：

```
interface Shape {
  onpaint(ctx: CanvasRenderingContext2D): void
}
```

类型	View	Controllers
QPaintDoc	onpaint(ctx)	addShape(shape)
QLine	onpaint(ctx)	new QLine(pt1, pt2, style)
QRect		new QRect(rect, style)
QEllipse		new QEllipse(x, y, radiusX, radiusY, style)
QPath		new QPath(points, close, style)
QLineStyle	-	new QLineStyle(width, color)

这个表列出的是 Model 和 View、Controllers 的耦合关系：Model 都为它们提供了什么？可以看出，View 层当前对 Model 层除了绘制（onpaint），没有其他任何需求。而各个 Controller，对 Model 的需求看起来似乎方法数量不少，但是实质上目的也只有一个，那就是创建图形（addShape）。

我们再看 View 层。它的代码主要是一个 [index.htm](#) 文件和一个 [view.js](#) 文件。View 层只依赖 Model 层，并且只依赖一个 doc.onpaint 函数。所以我们将关注点放在 View 自身的功能。

View 层只有一个 QPaintView 类。我们将其功能分为了三类：属于 Model 层职责相关的，属于 View 自身职责相关的，以及为 Controller 层服务的，得到下表。

```
interface Controller {
  stop(): void
  onpaint(ctx: CanvasRenderingContext2D): void
}
```

类型	Model	View	Controllers
数据	doc: QPaintDoc	properties: { lineWidth: number lineColor: string } drawing: DOMElement	controllers: map[string]Controller
方法	-	invalidateRect(rect)	get currentKey() get lineStyle() getMousePos(event) registerController(name, ctrl) invokeController(name) stopController()
事件	-	onpaint(ctx)	onmousedown(event) onmousemove(event) onmouseup(event) ondblclick(event) onkeydown(event)

最后，我们来看 Controller 层。Controller 层的文件有很多，这还是一些 Controller 因为实现相近被合并到一个文件，如下所示。

Menu, PropSelectors, MousePosTracker : [accel/menu.js](#)

Create Path : [creator/path.js](#)

Create FreePath : [creator/freepath.js](#)

Create Line, Rect, Ellipse, Circle : [creator/rect.js](#)

Controller 位于 MVC 的最上层，我们对它的关注点就不再是它的规格本身，也没人去调用它的方法。所以我们把关注点放在了每个 Controller 都怎么用 Model 和 View 的。

我们列了个表，如下。注意 Controller 对事件（Event）的使用从 View 中单独列出来了。

类型	Event	Model	View
Menu	-	-	controllers: map[string]Controller get currentKey() invokeController(name)
PropSelectors	-	-	properties: { lineWidth: number lineColor: string }
MousePosTracker	onmousemove	-	getMousePos(event)
QPathCreator	onmousedown(event) onmousemove(event) onmouseup(event) ondblclick(event) onkeydown(event) onpaint(ctx)	new QPath(points, close, style) doc.addShape(shape)	getMousePos(event) invalidateRect(rect) registerController(name, ctrl)
QFreePathCreator	onmousedown(event) onmousemove(event) onmouseup(event) onkeydown(event) onpaint(ctx)	new QPath(points, close, style) doc.addShape(shape)	getMousePos(event) invalidateRect(rect) registerController(name, ctrl)
QRectCreator	onmousedown(event) onmousemove(event) onmouseup(event) onkeydown(event) onpaint(ctx)	new QLine(pt1, pt2, style) new QRect(rect, style) new QEllipse(x, y, radiusX, radiusY, style) doc.addShape(shape)	getMousePos(event) invalidateRect(rect) registerController(name, ctrl)

通过以上三张表对照着看，可以清晰看出 Model、View、Controllers 是怎么关联起来的。

改进版的画图程序

MVP 版本的画图程序，用着就会发现不好用，毕竟图形创建完就没法改了。所以我们打算做一个新版本出来，功能上有这样一些改进。

选择一个图形，允许删除、移动或者对其样式进行修改。

图形样式增加 fillColor (填充色)。

更加现代的交互范式：默认处于 ShapeSelector 状态，创建完图形后自动回到此状态。

选择图形后，界面上的当前样式自动更新为被选图形的样式。

怎么改我们的程序？

完整的差异对比，请参见：

下面，我们将详细讲解这些修改背后的思考。

我们先看 Model 层，新的规格见下表。

dom.js

```
interface Shape {
  onpaint(ctx: CanvasRenderingContext2D): void
  bound(): Rect
  hitTest(pt: Point): {hitCode: number, hitShape: Shape}
  setProp(key: string, val: any): void
  move(dx, dy: number): void
}
```

类型	View	Controllers
QPaintDoc	onpaint(ctx)	addShape(shape) deleteShape(shape) hitTest(pt)
QLine QRect QEllipse QPath	onpaint(ctx)	new QLine(pt1, pt2, style) new QRect(rect, style) new QEllipse(x, y, radiusX, radiusY, style) new QPath(points, close, style) bound() hitTest(pt) setProp(key, val) move(dx, dy)
QShapeStyle	new QShapeStyle(lineWidth, lineColor, fillColor)	setProp(key, val) clone()

为了方便大家理解，我们做了一个 Model 的 ChangeNotes 表格，如下：

类型	View	Controllers	修改说明
QPaintDoc	-	deleteShape(shape) hitTest(pt)	- 增加 hitTest (确定鼠标点中哪个图形)、deleteShape (删除某个图形), 都用于 QShapeSelector
QLine QRect QEllipse QPath	-	new QLine(pt1, pt2, style) new QRect(rect, style) new QEllipse(x, y, radiusX, radiusY, style) new QPath(points, close, style) bound() hitTest(pt) setProp(key, val) move(dx, dy)	- 构造函数 style 参数由 QLineStyle 改为 QShapeStyle - bound (求图形的外接矩形)、hitTest 用于选择图形 - setProp (修改图形样式的某个属性) - move (移动图形)
QShapeStyle	new QShapeStyle(lineWidth, lineColor, fillColor)	setProp(key, val) clone()	- QLineStyle 改名为 QShapeStyle - 属性 width、color 改名为 lineWidth、lineColor - 增加属性 fillColor (图形的填充色) - 增加 setProp、clone (克隆图形样式)

大部分是新功能的增加，不提。我们重点关注一个点：QLineStyle 改名为 QShapeStyle，且其属性 width、color 被改名为 lineWidth、lineColor。这些属于不兼容修改，相当于做了一次小重构。


重构关键是要及时处理，把控质量。尤其对 JavaScript 这种弱类型语言，重构的心智负担较大。为了保证质量仍然可控，最好辅以足够多的单元测试。

这也是我个人会更喜欢静态类型语言的原因，重构有任何遗漏，编译器会告诉你哪里漏改了。当然，这并不意味着单元测试可以省略，对每一门语言来说，自动化的测试永远是质量保障的重要手段。

话题回到图形样式。最初我们 new QLine、QRect、QEllipse、QPath 的时候，传入的最后一个参数是 QLineStyle，从设计上这是一次失误，这意味着后面这些构造还是都需要增加更多参数如 QFillStyle 之类。

把最后一个参数改为 QShapeStyle，这从设计上就完备了。后面图形样式就算有更多的演进，也会集中到 QShapeStyle 这一个类上。

当前 QShapeStyle 的数据结构是这样的：


 复制代码

```

1 class QShapeStyle {
2   lineWidth: number
3   lineColor: string
4   fillColor: string
5 }
```

那么，这是合理的么？未来潜在的演进是什么？

对需求演进的推演，关键是眼光看多远。当前各类 GDI 对 LineStyle、FillStyle 支持都非常丰富。所以如果作为一个实实在在要去迭代的画图程序来说，上面这个 QShapeStyle 必然还会面临一次重构。变成如下这个样子：

 复制代码

```
1 class QLineStyle {
2   width: number
3   color: string
4 }
5
6 class QFillStyle {
7   color: string
8 }
9
10 class QShapeStyle {
11   line: any
12   fill: any
13 }
```

为什么 QShapeStyle 里面的 line 不是 QLineStyle，fill 不是 QFillStyle，而是 any 类型？因为它们都只是简单版本的线型样式和填充样式。

举个例子，在 GDI 系统中，FillStyle 往往还可以是一张图片平铺，也可以是多个颜色渐变填充，这些都无法用 QFillStyle 来表示。所以这里的 QFillStyle 更好的叫法也许是 QSimpleFillStyle。

聊完了 Model 层，我们再来看 View 层。

[view.js](#)

```
interface Controller {
    stop(): void
    onpaint(ctx: CanvasRenderingContext2D): void
}
```

类型	Model	View	Controllers
数据	doc: QPaintDoc	style: QShapeStyle drawing: DOMELEMENT	controllers: map[string]Controller
方法	-	invalidateRect(rect)	get currentKey() get selection() set selection(shape) getMousePos(event) registerController(name, ctrl) invokeController(name) stopController() fireControllerReset()
事件	-	onpaint(ctx)	onmousedown(event) onmousemove(event) onmouseup(event) ondblclick(event) onkeydown(event) onSelectionChanged(old) onControllerReset()

View 层的变化不大。为了给大家更直观的感觉，我这里也列了一个 ChangeNotes 表格，如下：

类型	Model	View	Controllers	修改说明
数据	-	style: QShapeStyle	-	- 属性 properties 改名为 style
方法	-	-	get selection() set selection(shape) fireControllerReset()	- 删除了 get lineStyle(), 和 properties 统一为 style - 增加了 selection 读写 - fireControllerReset, 用于让创建图形的 Controller 完成或放弃图形创建时发出 onControllerReset 事件
事件	-	-	onSelectionChanged(old) onControllerReset()	- onSelectionChanged 在被选择的图形改变时发出 - onControllerReset (见 fireControllerReset 的说明)

其中，properties 改名为 style，以及删除了 get lineStyle(), 和 properties 统一为 style。这个和我上面说的 Model 层的小重构相关，并不是本次新版本的功能引起的。

所以 View 层真正的变化是两个：

引入了 selection，当前只能单选一个 shape；在 selection 变化时会发出 onSelectionChanged 事件；

引入了 onControllerReset 事件，它在 Controller 完成或放弃图形的创建时发出。

引入 selection 比较常规。View 变复杂了通常都会有 selection，唯一需要考虑的是 selection 会有什么样的变化，对于 Office 类程序，如果 selection 只允许是单 shape 这不太合理，但我们这里略过，不进行展开。

我们重点谈 onControllerReset 事件。

onControllerReset 事件是创建图形的 Controller（例如 QPathCreator、QRectCreator 等）发出，并由 Menu 这个 Controller 接收。


这就涉及了一个问题：类似情况还会有多少？以后是不是还会有更多的事件需要在 Controller 之间传递，需要 View 来中转的？

这个问题就涉及了 View 层事件机制的设计问题。和这个问题相关的有：

要不要支持任意的事件；

监听事件是支持单播还是多播？

从最通用的角度，肯定是支持任意事件、支持多播。比如我们定义一个 QEventManager 类，规格如下。

 复制代码

```
1 class QEventManager {
2     fire(eventName: string, params: ...any): void
3     addListener(eventName: string, handler: Handler): void
4     removeListener(eventName: string, handler: Handler): void
5 }
```



但是，View 的事件机制设定，需要在通用性与架构的可控性之平衡。一旦 View 聚合了这个 QEventManager，通用是通用了，但是 Controller 之间会有什么样的事件飞来飞去，就比较难去从机制上把控了。

代码即文档。如果能够用代码约束的事情，最好不要在文档中来约束。

所以，就算是我们底层实现 QEventManager 类，我个人也不倾向于在 View 的接口中直接将它暴露出去，而是定义更具体的 fireControllerReset、onControllerReset/offControllerReset 方法，让架构的依赖直观化。

具体代码看起来是这样的：

 复制代码

```
1 class QPaintView {
2     constructor() {
3         this._eventManager = new QEventManager()
4     }
5     onControllerReset(handler) {
6         this._eventManager.addListener("onControllerReset", handler)
7     }
8     offControllerReset(handler) {
9         this._eventManager.removeListener("onControllerReset", handler)
10    }
11    fireControllerReset() {
12        this._eventManager.fire("onControllerReset")
13    }
14 }
```

聊完了 View 层，我们接着聊 Controller 层。我们也把每个 Controller 怎么用 Model 和 View 列了个表，如下。

Menu, PropSelectors, MousePosTracker : [accel/menu.js](#)

ShapeSelector : [accel/select.js](#)

Create Path : [creator/path.js](#)

Create FreePath : [creator/freepath.js](#)

Create Line, Rect, Ellipse, Circle : [creator/rect.js](#)

类型	Event	Model	View
Menu	onControllerReset()	-	controllers: map[string]Controller get currentKey() invokeController(name)
PropSelectors	onSelectionChanged(old)	shape.style shape.setProp(key, val) shape.clone()	style: QShapeStyle get selection() invalidateRect(rect)
MousePosTracker	onmousemove	-	getMousePos(event)
QShapeSelector	onmousedown(event) onmousemove(event) onmouseup(event) onkeydown(event) onpaint(ctx)	doc.deleteShape(shape) doc.hitTest(pt) shape.move(dx, dy) shape.bound()	get selection() set selection(shape) getMousePos(event) invalidateRect(rect) registerController(name, ctrl)
QPathCreator	onmousedown(event) onmousemove(event) onmouseup(event) ondblclick(event) onkeydown(event) onpaint(ctx)	new QPath(points, close, style) doc.addShape(shape)	getMousePos(event) invalidateRect(rect) registerController(name, ctrl) fireControllerReset()
QFreePathCreator	onmousedown(event) onmousemove(event) onmouseup(event) onkeydown(event) onpaint(ctx)	new QPath(points, close, style) doc.addShape(shape)	getMousePos(event) invalidateRect(rect) registerController(name, ctrl) fireControllerReset()
QRectCreator	onmousedown(event) onmousemove(event) onmouseup(event) onkeydown(event) onpaint(ctx)	new QLine(pt1, pt2, style) new QRect(rect, style) new QEllipse(x, y, radiusX, radiusY, style) doc.addShape(shape)	getMousePos(event) invalidateRect(rect) registerController(name, ctrl) fireControllerReset()

内容有点多。为了更清楚地看到差异，我们做了 ChangeNotes 表格，如下：

类型	Event	Model	View	修改说明
Menu	onControllerReset()	-	-	- 引入了 v2 版本的切换 Controller 的范式，更接近现代的交互范式
PropSelectors	onSelectionChanged(old)	shape.style shape.setProp(key, val) style.clone()	style: QShapeStyle get selection() invalidateRect(rect)	- 这个 Controller 要比上一版本的复杂很多：之前只是修改 view 的 properties (现在是 style) 属性，以便于创建图形时引用。现在是改变它时还会作用于 selection (被选中的图形)，改变它的样式；而且，在 selection 改变时，会自动更新界面以反映被选图形的样式
MousePosTracker	-	-	-	- 无变化
QShapeSelector	onmousedown(event) onmousemove(event) onmouseup(event) onkeydown(event) onpaint(ctx)	doc.deleteShape(shape) doc.hitTest(pt) shape.move(dx, dy) shape.bound()	get selection() set selection(shape) getMousePos(event) invalidateRect(rect) registerController(name, ctrl)	- 完全新增的 Controller
QPathCreator	-	new QPath(points, close, style)	fireControllerReset()	- QPath 的 style 参数从 QLineStyle 变为 QShapeStyle - 完成或放弃图形创建时发出 onControllerReset 事件
QFreePathCreator	-	new QPath(points, close, style)	fireControllerReset()	- 同上
QRectCreator	-	new QLine(pt1, pt2, style) new QRect(rect, style) new QEllipse(x, y, radiusX, radiusY, style)	fireControllerReset()	- 同上

首先，Menu、QPathCreator、QFreePathCreator、QRectCreator 的变更，主要因为引入了新的交互范式导致，我们为此引入了 onControllerReset 事件。还有一个变化是 QLineStyle 变 QShapeStyle，这一点前面已经详细讨论，不提。

所以 Controller 层的变化其实主要是两个。

其一，PropSelectors。这个 Controller 要比上一版本的复杂很多：之前只是修改 View 的 properties (现在是 style) 属性，以便于创建图形时引用。现在是改变它时还会作用于 selection (被选中的图形)，改变它的样式；而且，在 selection 改变时，会自动更新界面以反映被选图形的样式。

其二，QShapSelector。这是新增加的 Controller，支持选择图形，支持删除、移动被选择的图形。

通过这次的需求迭代我们可以看出，目前 Model、View、Controller 的分工，可以使需求的分解非常正交。

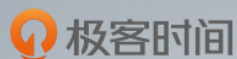
Model 只需要考虑需求导致的数据结构演进，并抽象出足够自然的业务接口。View 层非常稳定，主要起到各类角色之间的桥接作用。Controller 层每个 Controller 各司其职，彼此之间不会受到对方需求的干扰。

结语

今天我们结合“画图”程序重新梳理了一遍 MVC 架构。并且我们更进一步，通过对画图程序进行一次需求演进，来观察 MVC 架构各个角色对需求变更的敏感性。需要再次强调的是，虽然我们基于 Web 开发，但是我们当前给出的画图程序本质上还是单机版的。

如果你对今天的内容有什么思考与解读，欢迎给我留言，我们一起讨论。下一讲我们将继续实战一个联网版本的画图程序。

如果你觉得有所收获，也欢迎把文章分享给你的朋友。感谢你的收听，我们下期再见。




许式伟的架构课

从源头出发，带你重新理解架构设计

许式伟
七牛云 CEO



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 26 | 实战（一）：怎么设计一个“画图”程序？

下一篇 28 | 实战（三）：怎么设计一个“画图”程序？



有铭

2019-07-23

老师，我知道你这章讲了很厉害的东西，但是我也只能和上面某人说的一样，说一声：“不明觉厉”。原因不在别的，抽象现实需要对现实有很长时间的切身体会，必须要经历痛苦，有痛彻灵魂，深刻的总结，才能体会到解决方案的甜美。这里的大部分人，包括我在内，没有在实践里经历“从头构建一个画图”程序。所以对于这章的知识，看得懂，但是难有感悟，换个场景，就使不出来

展开 ▾

作者回复: 理解，我想一下怎么才能说得更明白。这里面每一个角色的分解是有非常明确的套路可循，遵循文章中说的几个要点即可。



7



许童童

2019-07-23

每一个字都看得懂，但连到一起就看不懂了。

展开 ▾

作者回复: 明白。这些反馈对我很有收益。



2



Being

2019-07-27

实战一二的例子套用前面的知识点，对MVC的理解慢慢清晰了。更重要的是在实战的学习中也结合现在自己手头的项目进行思考。现在在做地图方面结合图元的绘制，目前的平台是Qt。我负责的是图元模块，即要对图元抽象并且使用Qt的绘制方法，当然，对于Qt的部分是严格控制在绘制部分的，也是考虑了跨平台的因素。图元部分其实是抽象了接口的，对于上层的调用完全不用关心底层绘制是用的什么平台。整个项目也是明确了几个模块...

展开 ▾

作者回复: 我们这个例子其实controller都是插件，只是在init application的时候创建一下就完了，是不是实现成标准plugin机制不是关键点



1



3k

2019-07-26

MVC相对简单，可以讲讲微服务模型之类的架构么？

展开

作者回复: 这个在下一章，服务端架构



1



Taozi

2019-07-24

多看几遍代码，先看v26分支，已经明朗很多了。谢谢。



1



Charles

2019-07-24

许老师，我反复读了好久还是有点懵懂，可能要先去读下完整的源码再回来读可能好一些？

作者回复: 最好能够先大致看一下代码



1



我的腿腿

2019-07-23

最近在看GoF的设计模式，里面也是用图形界面做例子引入某某模式，和作者的不谋而合，不过还是太抽象了！感觉在爬一座充满荆棘的山

展开

作者回复: 本例中什么地方没看懂？



Aaron Cheung

2019-07-28

补打卡 27

展开 ∨



DemonLee

2019-07-27

我也没怎么看懂，估计是不会前端，看的时候我一直在告诉自己：老师是要告诉我们，解耦，解耦，解耦！

作者回复：嗯，后面可以看一个不解耦的版本



刘宗尧

2019-07-25

能类比一下移动的操作系统吗？android或者ios

展开 ∨

作者回复：是说它们有什么差异？站在我们架构分析的纬度，两者大同小异



梦里

2019-07-24

好文章需要多看几遍。

展开 ∨



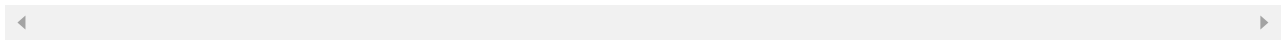
Charles

2019-07-24

另外有个建议，文中的类、方法、事件这些用uml图或自定义流程图之类的表达会不会更

直观一些？

作者回复: 主要我们的侧重点是子系统（或模块）之间的耦合，所以并没有把关注点放在流程上

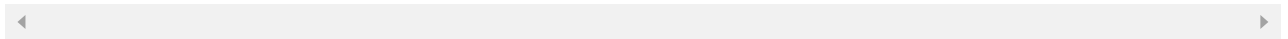


吴

2019-07-23

有深度，模模糊糊懂，具体细节不太明白，没做过cs端的估计也看不懂

作者回复: 其实这个程序虽然是b/s结构的，但是实质上是单机版本的，只需要看js代码即可



筱恕

2019-07-23

感觉这一章有几本书的知识量

展开 ∨



Jian

2019-07-23

代码面向扩展，面向对象，高内聚低耦合。设计者需要很强的抽象思维能力，才能设计出艺术品般的代码。开发者需要更多有效锤炼才能掌握这样的能力。

展开 ∨

