如何将 HTML5 性能发挥到极致

2016-05-30 分类：WEB开发、编程开发、首页精华 1人评论 来源：极客头条

HTML5作为新兴领域越来越热。然而在移动设备硬件性能弱于PC的背景下，对性能的需求显得更为重要，而HTML5性能优化前与优化后有着极大的差别，如何优化才能提高性能，对此熟知的人很少。本文以LayaAir引擎为例，通过代码示例详细阐述如何利用引擎对HTML5作出性能的极致优化。

主题包括：

- 代码执行基本原理
- 基准测试
- 内存优化
- 图形渲染性能
- 减少CPU使用量
- 其他优化策略

第1节：代码执行基本原理

LayaAir引擎支持AS3、TypeScript、JavaScript三种语言开发，然而无论是采用哪种开发语言，最终执行的都是JavaScript代码。所有看到的画面都是通过引擎绘制出来的，更新频率取决于开发者指定的FPS，例如指定帧频率为60FPS，则运行时每个帧的执行时间为六十分之一秒，所以帧速越高，视觉上感觉越流畅，60帧是满帧。

由于实际运行环境是在浏览器中，因此性能还取决于JavaScript解释器的效率，指定的FPS帧速在低性能解释器中可能不会达到，所以这部分不是开发者能够决定的，开发者能作的是尽可能通过优化，在低端设备或低性能浏览器中，提升FPS帧速。

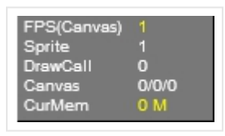
LayaAir引擎在每帧都会重绘，在性能优化时，除了关注每帧执行逻辑代码带来的CPU消耗，还需要注意每帧调用绘图指令的数量以及GPU的纹理提交次数。

第2节：基准测试

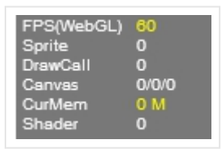
LayaAir引擎内置的性能统计工具可用于基准测试，实时检测当前性能。开发者可以使用laya.utils.Stat类，通过Stat.show() 显示统计面板。具体编写代码如下例所示：

```
Stat.show(0,0); //AS3的面板调用写法
Laya.Stat.show(0,0); //TS与JS的面板调用写法
```

Canvas渲染的统计信息：



WebGL渲染的统计信息：



统计参数的意义：

FPS：

每秒呈现的帧数(数字越高越好)。
使用canvas渲染时，描述字段显示为FPS(Canvas)，使用WebGL渲染时，描述字段显示为FPS(WebGL)。

Sprite：

渲染节点数量（数字越低越好）。
Sprite统计所有渲染节点（包括容器），这个数字的大小会影响引擎节点遍历，数据组织和渲染的次数。

DrawCall：

DrawCall在canvas和WebGL渲染下代表不同的意义（越少越好）。
Canvas下表示每帧的绘制次数，包括图片、文字、矢量图。尽量限制在100之下。
WebGL下表示渲染提交批次，每次准备数据并通知GPU渲染绘制的过程称为1次DrawCall，在每1次DrawCall中除了在通知GPU的渲染上比较耗时之外，切换材质与shader也是非常耗时的操作。 DrawCall的次数是决定性能的重要指标，尽量限制在100之下。

Canvas：

三个数值 —— 每帧重绘的画布数量 / 缓存类型为“normal”类型的画布数量 / 缓存类型为“bitmap”类型的画布数量”。
CurMem：仅限WebGL渲染，表示内存与显存占用（越低越好）。
Shader：仅限WebGL渲染，表示每帧Shader提交次数。

无论是Canvas模式还是WebGL模式，我们都需要重点关注DrawCall，Sprite，Canvas这三个参数，然后针对性地进行优化。（参见“图形渲染性能”）

第3节：内存优化

热门栏目订阅
Java RSS 微博
程序员 RSS 微博
Android RSS
PHP RSS
JavaScript RSS
Linux RSS

热门文章
编程不需要天赋和激情
C# Lambda表达式的前世今生
22 个 Android Studio 优秀插件汇总
AngularJS 样式指南介绍
Java 远程通讯技术及原理分析
游戏编程十年总结
程序员眼中的古典名画
记一次 Google 面试经历
80 多个 Linux 系统管理员的监控工具
对优秀程序员的思考

职场人生
编程不需要天赋和激情
入门软件工程师所面临的5个挑战
程序员编程的 7 + 1 条小贴士
假如程序员生活在童话里...
一个32岁入门的70后程序员给我的启示
程序员每天都在使用的6个惊讶的软技能
一流程序员完全可以有编程之外的生活
回顾15年程序员生涯，我总结的7点经验
为什么开源可以提高程序员的编程技能？
程序员的走与留？

相关文章
HTML5超炫酷粒子效果的进度条
设计师必看的10个HTML5动画工具
HTML5 Google电吉他 可用键盘弹奏
HTML5 3D衣服摇摆动画特效
HTML5 3D书本翻页动画

关注我们的微博
程序员俱乐

HTML5和JS实现的切水果游戏

逼真的HTML5 3D水波动画 可多视角浏览

多视角3D可旋转的HTML5 Logo动画

对象池

对象池，涉及到不断重复使用对象。在初始化应用程序期间创建一定数量的对象并将其存储在一个池中。对一个对象完成操作后，将该对象放回到池中，在需要新对象时可以对其进行检索。
由于实例化对象成本很高，使用对象池重用对象可减少实例化对象的需求。还可以减少垃圾回收器运行的机会，从而提高程序的运行速度。

以下代码演示使用

Laya.utils.Pool：

```
ar SPRITE_SIGN = 'spriteSign';
var sprites = [];
function initialize()
{
        for (var i = 0; i < 1000; i++)
        {
                var sp = Pool.getItemByClass(SPRITE_SIGN, Sprite)
                sprites.push(sp);
                Laya.stage.addChild(sp);
        }
}
initialize();
```

在initialize中创建大小为1000的对象池。

以下代码在当单击鼠标时，将删除显示列表中的所有显示对象，并在以后的其他任务中重复使用这些对象：

```
Laya.stage.on("click", this, function()
{
        var sp;
        for(var i = 0, len = sprites.length; i < len; i++)
        {
                sp = sprites.pop();
                Pool.recover(SPRITE_SIGN, sp);
                Laya.stage.removeChild(sp);
        }
});
```

调用Pool.recover后，指定的对象会被回收至池内。

使用Handler.create

在开发过程中，会经常使用Handler来完成异步回调。Handler.create使用了内置对象池管理，因此在使用Handler对象时应使用Handler.create来创建回调处理器。以下代码使用Handler.create创建加载的回调处理器：

```
Laya.loader.load(urls, Handler.create(this, onAssetLoaded));
```

在上面的代码中，回调被执行后Handler将会被对象池收回。此时，考虑如下代码会发生什么事：

```
Laya.loader.load(urls, Handler.create(this, onAssetLoaded), Handler.create(this, onLoadin
```

在上面的代码中，使用Handler.create返回的处理器处理progress事件。此时的回调执行一次之后就被对象池回收，于是progress事件只触发了一次，此时需要将四个名为once的参数设置为false：

```
Laya.loader.load(urls, Handler.create(this, onAssetLoaded), Handler.create(this, onLoadin
```

释放内存

JavaScript运行时无法启动垃圾回收器。要确保一个对象能够被回收，请删除对该对象的所有引用。Sprite提供的destory会帮助设置内部引用为null。

例如，以下代码确保对象能够被作为垃圾回收：

```
var sp = new Sprite();
sp.destroy();
```

当对象设置为null，不会立即将其从内存中删除。只有系统认为内存足够低时，垃圾回收器才会运行。内存分配（而不是对象删除）会触发垃圾回收。

垃圾回收期间可能占用大量CPU并影响性能。通过重用对象，尝试限制使用垃圾回收。此外，尽可能将引用设置为null，以便垃圾回收器用较少时间来查找对象。有时（比如两个对象相互引用），无法同时设置两个引用为null，垃圾回收器将扫描无法被访问到的对象，并将其清除，这会比引用计数更消耗性能。

资源卸载

游戏运行时总会加载许多资源，这些资源在使用完成后应及时卸载，否则一直残留在内存中。

下例演示加载资源后对比资源卸载前和卸载后的资源状态：

```
var assets = [];
assets.push("res/apes/monkey0.png");
assets.push("res/apes/monkey1.png");
assets.push("res/apes/monkey2.png");
assets.push("res/apes/monkey3.png");

Laya.loader.load(assets, Handler.create(this, onAssetsLoaded));

function onAssetsLoaded()
{
        for(var i = 0, len = assets.length; i < len; ++i)
        {
                var asset = assets[i];
                console.log(Laya.loader.getRes(asset));
                Laya.loader.clearRes(asset);
                console.log(Laya.loader.getRes(asset));
```

```
            }
        }
```

关于滤镜、遮罩

尝试尽量减少使用滤镜效果。将滤镜（BlurFilter和GlowFilter）应用于显示对象时，运行时将在内存中创建两张位图。其中每个位图的大小与显示对象相同。将第一个位图创建为显示对象的栅格化版本，然后用于生成应用滤镜的另一个位图：

应用滤镜时内存中的两个位图

当修改滤镜的某个属性或者显示对象时，内存中的两个位图都将更新以创建生成的位图，这两个位图可能会占用大量内存。此外，此过程涉及CPU计算，动态更新时将会降低性能（参见“图形渲染性能 – 关于cacheAs）。

ColorFiter在Canvas渲染下需要计算每个像素点，而在WebGL下的GPU消耗可以忽略不计。

最佳的做法是，尽可能使用图像创作工具创建的位图来模拟滤镜。避免在运行时中创建动态位图，可以帮助减少CPU或GPU负载。特别是一张应用了滤镜并且不会在修改的图像。

第4节：图形渲染性能

优化Sprite

1.尽量减少不必要的层次嵌套，减少Sprite数量。
2.非可见区域的对象尽量从显示列表移除或者设置visible=false。
3.对于容器内有大量静态内容或者不经常变化的内容（比如按钮），可以对整个容器设置cacheAs属性，能大量减少Sprite的数量，显著提高性能。如果有动态内容，最好和静态内容分开，以便只缓存静态内容。
4.Panel内，会针对panel区域外的直接子对象（子对象的子对象判断不了）进行不渲染处理，超出panel区域的子对象是不产生消耗的。

优化DrawCall

1.对复杂静态内容设置cacheAs，能大量减少DrawCall，使用好cacheAs是游戏优化的关键。
2.尽量保证同图集的图片渲染顺序是挨着的，如果不同图集交叉渲染，会增加DrawCall数量。
3.尽量保证同一个面板中的所有资源用一个图集，这样能减少提交批次。

优化Canvas

在对Canvas优化时，我们需要注意，在以下场合不要使用cacheAs：

1.对象非常简单，比如一个字或者一个图片，设置cacheAs=bitmap不但不提高性能，反而会损失性能。
2.容器内有经常变化的内容，比如容器内有一个动画或者倒计时，如果再对这个容器设置cacheAs=bitmap，会损失性能。

可以通过查看Canvas统计信息的第一个值，判断是否一直在刷新Canvas缓存。

关于cacheAs

设置cacheAs可将显示对象缓存为静态图像，当cacheAs时，子对象发生变化，会自动重新缓存，同时也可以手动调用reCache方法更新缓存。 建议把不经常变化的复杂内容，缓存为静态图像，能极大提高渲染性能，cacheAs有”none”，”normal”和”bitmap”三个值可选。

1. 默认为”none”，不做任何缓存。
   2.当值为”normal”时，canvas下进行画布缓存，webgl模式下进行命令缓存。
   3.当值为”bitmap”时，canvas下进行依然是画布缓存，webGL模式下使用renderTarget缓存。这里需要注意的是，webGL下renderTarget缓存模式有2048大小限制，超出2048会额外增加内存开销。另外，不断重绘时开销也比较大，但是会减少drawcall，渲染性能最高。 webGL下命令缓存模式只会减少节点遍历及命令组织，不会减少drawcall，性能中等。

设置cacheAs后，还可以设置staticCache=true以阻止自动更新缓存，同时可以手动调用reCache方法更新缓存。

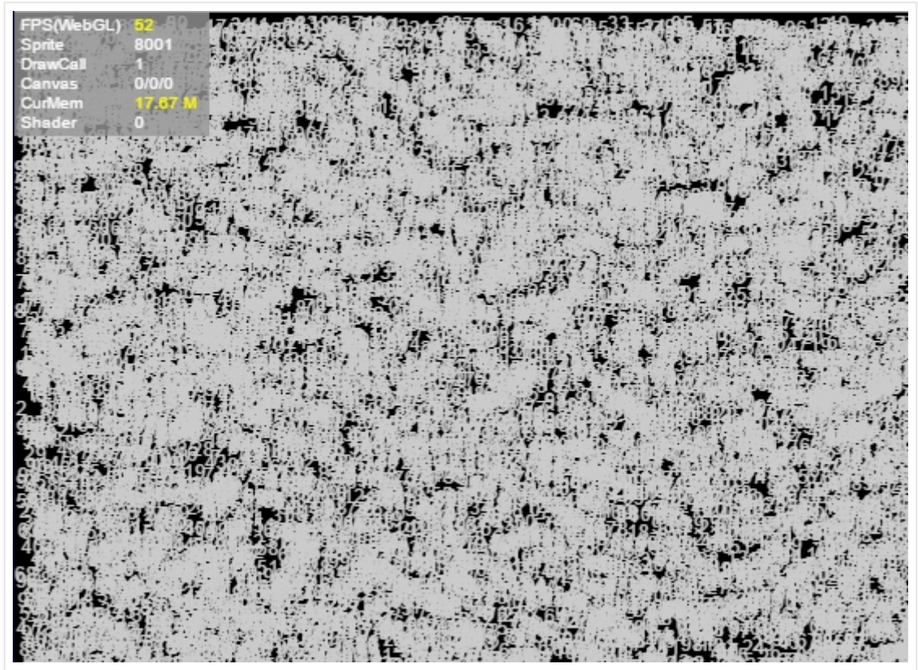
cacheAs主要通过两方面提升性能。一是减少节点遍历和顶点计算；二是减少drawCall。善用cacheAs将是引擎优化性能的利器。

下例绘制10000个文本：

```
Laya.init(550, 400, Laya.WebGL);
Laya.Stat.show();

var textBox = new Laya.Sprite();

var text;
for (var i = 0; i < 10000; i++)
{
    text = new Laya.Text();
    text.text = (Math.random() * 100).toFixed(0);
    text.color = "#CCCCCC";

    text.x = Math.random() * 550;
```

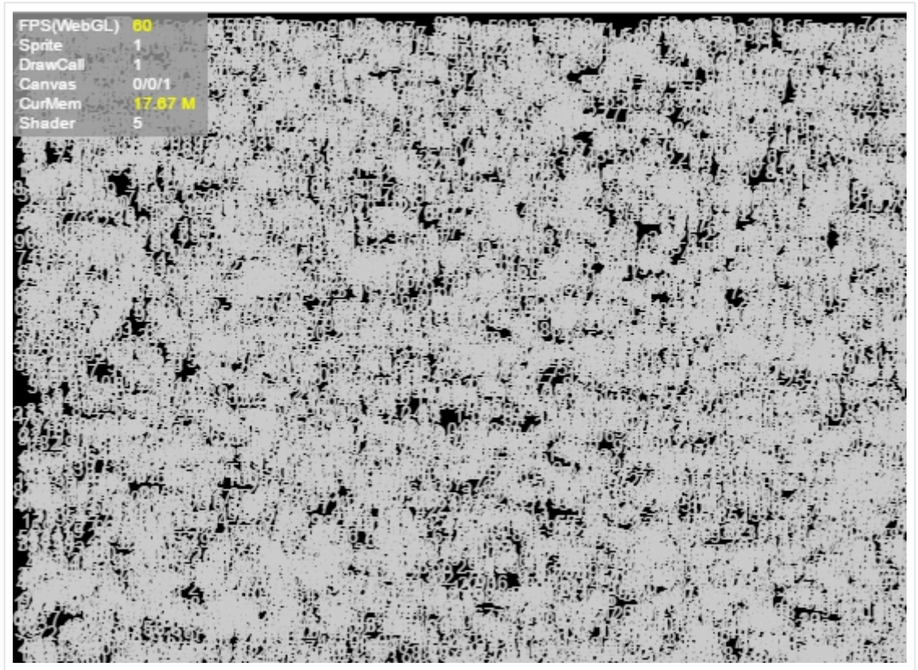
```
    text.y = Math.random() * 400;

    textBox.addChild(text);
}

Laya.stage.addChild(textBox);
```

下面是笔者电脑上的运行时截图，FPS稳定于52上下。



当我们对文字所在的容器设置为cacheAs之后，如下面的例子所示，性能获得较大的提升，FPS达到到了60帧。

```
// …省略其他代码… var textBox = new Laya.Sprite();
textBox.cacheAs = "bitmap"; // …省略其他代码…
```



文字描边

在运行时，设置了描边的文本比没有描边的文本多调用一次绘图指令。此时，文本对CPU的使用量和文本的数量成正比。因此，尽量使用替代方案来完成同样的需求。

对于几乎不变动的文本内容，可以使用cacheAs降低性能消耗，参见“图形渲染性能 – 关于cacheAs”。

对于内容经常变动，但是使用的字符数量较少的文本域，可以选择使用位图字体。

跳过文本排版，直接渲染

大多数情况下，很多文本都不需要复杂的排版，仅仅简单地显示一行字。为了迎合这一需求，Text提供的名为changeText的方法可以直接跳过排版。

```
var text = new Text();
text.text = "text";
Laya.stage.addChild(text);
//后面只是更新文字内容，使用changeText能提高性能
text.changeText("text changed.");
```

Text.changeText会直接修改绘图指令中该文本绘制的最后一条指令，这种前面的绘图指令依旧存在的行为会导致changeText只使用于以下情况：

文本始终只有一行。

文本的样式始终不变（颜色、粗细、斜体、对齐等等）。

即使如此，实际编程中依旧会经常使用到这样的需要。

第5节：减少CPU使用量

减少动态属性查找

JavaScript中任何对象都是动态的，你可以任意地添加属性。然而，在大量的属性里查找某属性可能很耗时。如果需要频繁使用某个属性值，可以使用局部变量来保存它：

```
function foo()
{
    var prop = target.prop;
    // 使用prop
    process1(prop);
    process2(prop);
    process3(prop);
}
```

计时器

LayaAir提供两种计时器循环来执行代码块。

1. Laya.timer.frameLoop执行频率依赖于帧频率，可通过Stat.FPS查看当前帧频。
2. Laya.timer.loop执行频率依赖于参数指定时间。

当一个对象的生命周期结束时，记得清除其内部的Timer：

```
Laya.timer.frameLoop(1, this, animateFrameRateBased);
Laya.stage.on("click", this, dispose);
function dispose()
{
    Laya.timer.clear(this, animateFrameRateBased);
}
```

获取显示对象边界的做法

在相对布局中，很经常需要正确地获取显示对象的边界。获取显示对象的边界也有多种做法，而其间差异很有必要知道。

1.使用getBounds/ getGraphicBounds。、

```
var sp = new Sprite();
sp.graphics.drawRect(0, 0, 100, 100, "#FF0000");
var bounds = sp.getGraphicBounds();
Laya.stage.addChild(sp);
```

getBounds可以满足多数多数需求，但由于其需要计算边界，不适合频繁调用。

2.设置容器的autoSize为true。

```
var sp = new Sprite();
sp.autoSize = true;
sp.graphics.drawRect(0, 0, 100, 100, "#FF0000");
Laya.stage.addChild(sp);
```

上述代码可以在运行时正确获取宽高。autoSize在获取宽高并且显示列表的状态发生改变时会重新计算（autoSize通过getBoudns计算宽高）。所以对拥有大量子对象的容器应用autoSize是不可取的。如果设置了size，autoSize将不起效。

使用loadImage后获取宽高：

```
var sp = new Sprite();
sp.loadImage("res/apes/monkey2.png", 0, 0, 0, 0, Handler.create(this, function()
{
        console.log(sp.width, sp.height);
}));
Laya.stage.addChild(sp);
```

loadImage在加载完成的回调函数触发之后才可以正确获取宽高。

3.直接调用size设置：

```
Laya.loader.load("res/apes/monkey2.png", Handler.create(this, function()
{
        var texture = Laya.loader.getRes("res/apes/monkey2.png");
        var sp = new Sprite();
        sp.graphics.drawTexture(texture, 0, 0);
        sp.size(texture.width, texture.height);
        Laya.stage.addChild(sp);
}));
```

使用Graphics.drawTexture并不会自动设置容器的宽高，但是可以使用Texture的宽高赋予容器。毋庸置疑，这是最高效的方式。

注：getGraphicsBounds用于获取矢量绘图宽高。

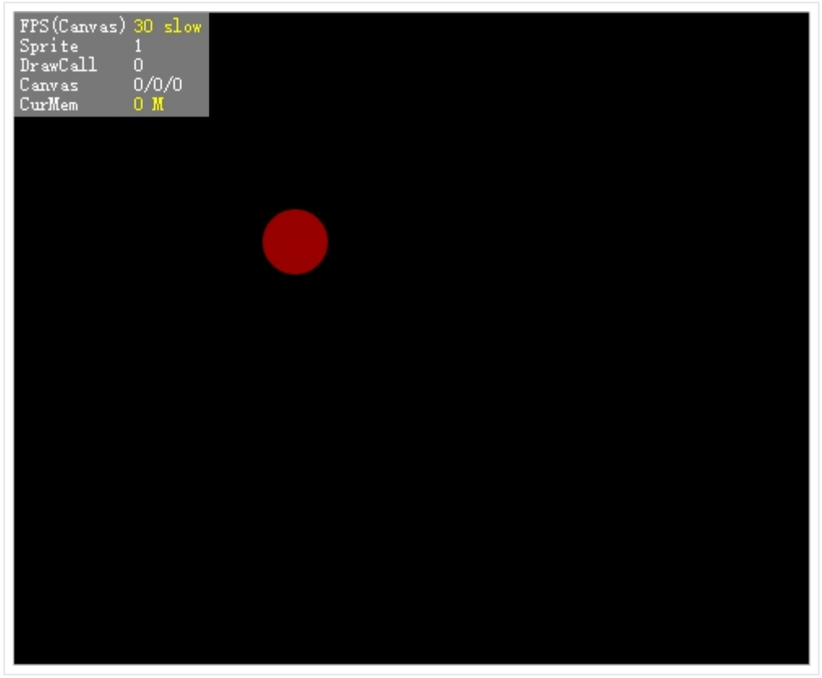
根据活动状态改变帧频

帧频有三种模式，Stage.FRAME_SLOW维持FPS在30；Stage.FRAME_FAST维持FPS在60；Stage.FRAME_MOUSE则选择性维持FPS在30或60帧。

有时并不需要让游戏以60FPS的速率执行，因为30FPS已经能够满足多数情况下人类视觉的响应，但是鼠标交互时，30FPS可能会造成画面的不连贯，于是Stage.FRAME_MOUSE应运而生。

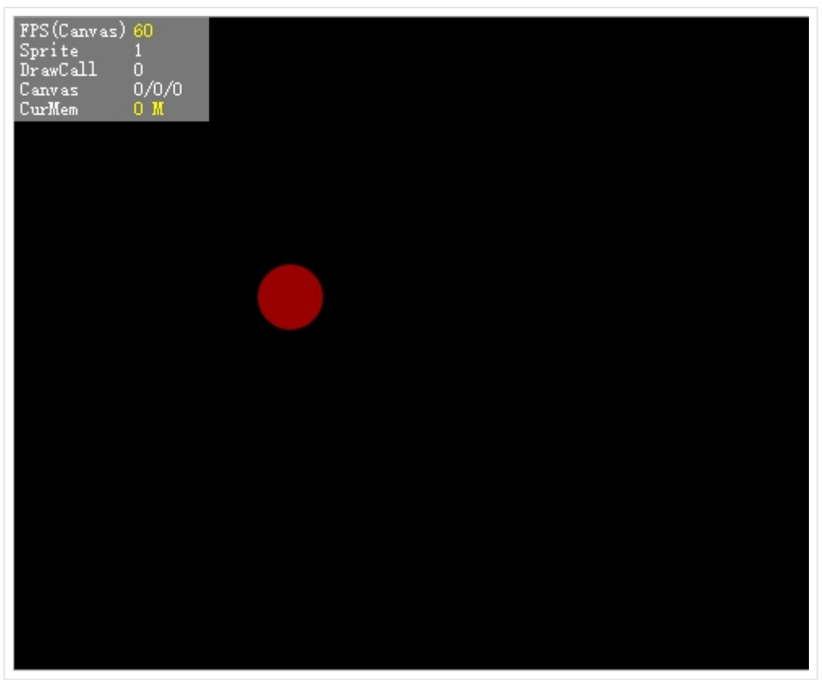
下例展示以Stage.FRAME_SLOW的帧率，在画布上移动鼠标，使圆球跟随鼠标移动：

```
Laya.init(Browser.width, Browser.height);
Stat.show();
Laya.stage.frameRate = Stage.FRAME_SLOW;

var sp = new Sprite();
sp.graphics.drawCircle(0, 0, 20, "#990000");
Laya.stage.addChild(sp);

Laya.stage.on(Event.MOUSE_MOVE, this, function()
{
        sp.pos(Laya.stage.mouseX, Laya.stage.mouseY);
});
```



此时FPS显示30，并且在鼠标移动时，可以感觉到圆球位置的更新不连贯。设置Stage.frameRate为Stage.FRAME_MOUSE：

```
Laya.stage.frameRate = Stage.FRAME_MOUSE;
```



此时在鼠标移动后FPS会显示60，并且画面流畅度提升。在鼠标静止2秒不动后，FPS又会恢复到30帧。

使用callLater

callLater使代码块延迟至本帧渲染前执行。如果当前的操作频繁改变某对象的状态，此时可以考虑使用callLater，以减少重复计算。

考虑一个图形，对它设置任何改变外观的属性都将导致图形重绘：

```
var rotation = 0,
        scale = 1,
        position = 0;

function setRotation(value)
{
        this.rotation = value;
        update();
}

function setScale(value)
{
        this.scale = value;
        update();
}

function setPosition(value)
{
        this.position = value;
        update();
}

function update()
{
        console.log('rotation: ' + this.rotation + '\tscale: ' + this.scale + '\tposition
}
```

调用以下代码更改状态：

```
setRotation(90); setScale(2); setPosition(30);
```

控制台的打印结果是

rotation: 90 scale: 1 position: 0
rotation: 90 scale: 2 position: 0
rotation: 90 scale: 2 position: 30

update被调用了三次，并且最后的结果是正确的，但是前面两次调用都是不需要的。

尝试将三处update改为：

```
Laya.timer.callLater(this, update);
```

此时，update只会调用一次，并且是我们想要的结果。

图片/图集加载

在完成图片/图集的加载之后，引擎就会开始处理图片资源。如果加载的是一张图集，会处理每张子图片。如果一次性处理大量的图片，这个过程可能会造成长时间的卡顿。

在游戏的资源加载中，可以将资源按照关卡、场景等分类加载。在同一时间处理的图片越少，当时的游戏响应速度也会更快。在资源使用完成后，也可以予以卸载，释放内存。

第6节：其他优化策略

1.减少粒子使用数量，在移动平台Canvas模式下，尽量不用粒子；

2.在Canvas模式下，尽量减少旋转，缩放，alpha等属性的使用，这些属性会对性能产生消耗。（在WebGL模式可以使用）；

3.不要在timeloop里面创建对象及复杂计算；

4.尽量减少对容器的autoSize的使用，减少getBounds()的使用，因为这些调用会产生较多计算；

5.尽量少用try catch的使用，被try catch的函数执行会变得非常慢；