思考

背景

1. 这是一个基于`H5 Canvas`的一个任务需求
2. 项目开发需要用到`canvas`，技术选型时考虑使用`ECharts`的`zrender`开发的，但是由于在wpf内嵌入网页链接打开的时候遇到无法解决的BUG（IE内核）
3. 决定模仿`zrender-1.0`版本 开发一个`canvas`小插件

遇到的问题

1. 如何将图形按照层次去加载
2. 以及解决图形（包括图片）异步加载的问题
3. 由于`canvas`机制是一般情况下 后面的图片会覆盖在前者

代码分析

> 装载递归函数的链栈

```
FUNC_LINK: [],
```

> 递归函数

```
_next: function(start) {
  var func_link = this.FUNC_LINK,
    func = func_link[start],
    that = this,
    arg = arguments;

  if(start >= func_link.length) return false;
  if(typeof func === 'function') {
  $.when(func())
    .done(function() {
      arg.callee.bind(that, ++start)(); // 严格模式下是报错的
      // that._next(++start, func_link);
    })
    .fail(function(errOpt) {
      if(typeof errOpt == 'undefined') return
      $.CanvasUtil._dealError(errOpt, that.container);
    });
  }
}
```

> 更新画布元素，根据缓存数据生产对应的overlay函数链

```
_update: function() {
   var that = this,
       ctx = this.ctx,

       width = this.canvas.width,
```

```
        height = this.canvas.height,
        func_link = []; // 函数链

    this.FUNC_LINK = [];

    var dataStorage = this.DataStorage;
    var p = this.params,
        curX = p.curX,
        curY = p.curY;
    dataStorage.forEach(function(key, i) {
      key && key.forEach(function(key, j) {
        var k = key,
            type = k.type,
            d = k; // 传入的数据源
        switch(type) {
          case 'image':
            func_link.push($.CanvasUtil._drawImage.bind(that, d));
            break;
          case 'circle':
            func_link.push($.CanvasUtil._drawCircle.bind(that, d));
            break;
          case 'polygon':
            func_link.push($.CanvasUtil._drawPolygon.bind(that, d));
            break;
          case 'line':
            func_link.push($.CanvasUtil._drawLine.bind(that, d));
            break;
          case 'polyline':
            func_link.push($.CanvasUtil._drawPolyline.bind(that, d));
            break;
          case 'rectangle':
            func_link.push($.CanvasUtil._drawRect.bind(that, d));
            break;
          default:
            break;
        }
      })
    });
    this.FUNC_LINK = func_link;
    this._draw();
  }
```

开启递归调用 函数链

```
_draw: function() {
  var that = this,
      ctx = this.ctx;

  $.CanvasUtil._clear.bind(this)();

  that._transform(ctx);

  that._next(0); //
}
```

业务处理 -- 处理overlay

```
_drawXX: function(D) {
  var that = this,
      i = D._data,
      ctx = this.ctx,
      dtd = $.Deferred();

  function _action() {
    ctx.save();
    // 操作
    ctx.restore();

    // 更改状态
    dtd.resolve();
  }
  return dtd;
}
```

遇到的坑

> Maximum call stack size exceeded
>
> 在调用_next递归函数的时候，由于调用栈缓存太多环境变量，导致内存溢出

改善 -- 尾递归函数优化

1. _next函数是尾递归优化的实现，重点就在于状态变量active。默认情况下，这个变量是不激活的
2. 一旦进入尾递归优化的过程，这个变量就激活了。然后，因为!active不成立，所以每一轮递归sum返回的都是undefined，所以就避免了递归执行
3. accumulated数组存放每一轮sum执行的参数，总是有值的，这就保证了accumulator函数内部的while循环总是会执行
4. 巧妙地将“递归”改成了“循环”，而后一轮的参数会取代前一轮的参数，保证了调用栈只有一层

```
_next: function(f) {
  var value;
  var active = false;
  var accumulated = [];

  return function accumulator() {
    accumulated.push(arguments);
```

```
    if(!active) {
      active = true;
      while(accumulated.length) {
        value = f.apply(this, accumulated.shift());
      }
      active = false;
      return value;
    }
  };
}
```

```
_draw: function() {
  var that = this;
  var fn = that._next(function(x, y, func_link) {
    if(y > -1) {
      var func = func_link[x];

      if(x >= func_link.length) return false;
      if(typeof func === 'function') {
        $.when(func())
        .done(function() {
          console.log(x, y);
          return fn(x + 1, y - 1, func_link);
        })
        .fail(function(errOpt) {
          if(typeof errOpt == 'undefined') return
          $.CanvasUtil._dealError(errOpt, that.container);
        });
      }
    } else {
      return x
    }
  });
  var func_link = this.FUNC_LINK;
  var len = func_link.length;
  fn(0, len-1, func_link)
}
```