彻底理解LEAFLET之 L.CRS + 搞定LEAFLET多坐标系拓展

一、抛砖引玉

对于Leaflet创建map。总有一个很重要头疼的属性– `crs` :

```
this.map = L.map("view-map", {
     zoomControl: false,
     attributionControl: false,
     crs: L.CRS.EPSG3857,
     minZoom: 1,
     maxZoom: 20,
 }).setView([38.00315, 114.28898], 4);
```

对。就是这么一个简单的属性，却是整个leaflet瓦片加载的核心算法必然依赖项。当然，对于leaflet官档，自然而然的给出了一句忠告：

crs	CRS	L.CRS.EPSG3857	The Coordinate Reference System to use. Don't change this if you're not sure what it means.

Don’t change this if you’re not sure what it means.

嘿嘿。你这给了一个不要轻易去动，然后给死了一个坐标系 `L.CRS.EPSG3857` 。这怎么可能满足我等的需求嘛？如果我要支持地方坐标系呢？咋办？

二、进入主题

好奇的我们打开了leaflet的源码。发现对于CRS的定义就几个函数:

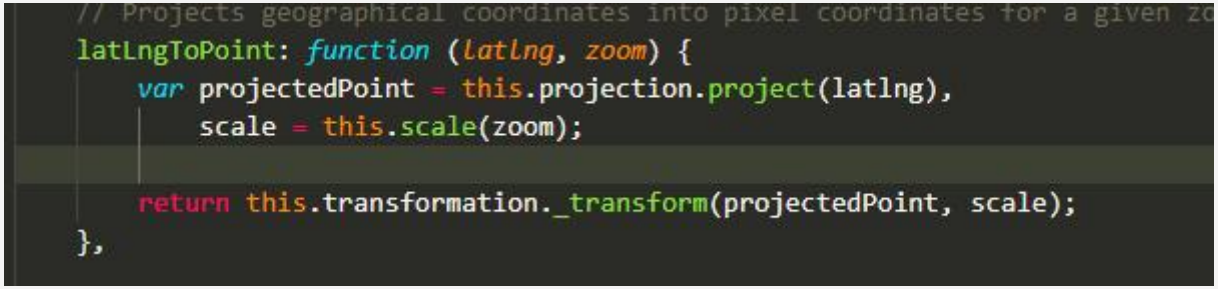
```
var CRS = {
    // @method latLngToPoint(latlng: LatLng, zoom: Number): Point
    // Projects geographical coordinates into pixel coordinates for a given zo
    latLngToPoint: function (latlng, zoom) {…
    },

    // @method pointToLatLng(point: Point, zoom: Number): LatLng
    // The inverse of `latLngToPoint`. Projects pixel coordinates on a given
    // zoom into geographical coordinates.
    pointToLatLng: function (point, zoom) {…
    },

    // @method project(latlng: LatLng): Point
    // Projects geographical coordinates into coordinates in units accepted fo
    // this CRS (e.g. meters for EPSG:3857, for passing it to WMS services).
    project: function (latlng) {…
    },

    // @method unproject(point: Point): LatLng
    // Given a projected coordinate returns the corresponding LatLng.
    // The inverse of `project`.
    unproject: function (point) {…
    },
```

没错。你看出来了，其实这玩意就是**提供了wgs84坐标系与目标坐标系之间转换的一种算法**。因为leaflet内部都是用经纬度坐标系表达的坐标，所以leaflet内部表达所用的坐标系都是 `wgs84` ，也就是 `EPSG:4326` 。所以啊。不管是计算瓦片行列号，计算瓦片最终在地图上的位置，都需要对应的坐标系的定义进行转换。L.CRS就是定义了这样的一种转换算法。我们来看一下 `L.CRS.latLngToPoint` 内部实现：



没错。调用了 `this.projection.project()` 这个函数。那 `this.projection` 的定义呢？没错。单独使用L.CRS这个是无法实现坐标转换的。这只是一个模板，是一个壳，真正做转换的其实是 `this.projection` 做的事情。于是我们往下看，Leaflet到底在这个 `L.CRS` 里做了什么文章：

```
var Earth = extend({}, CRS, {
    wrapLng: [-180, 180],

    // Mean Earth Radius, as recommended for use by
    // the International Union of Geodesy and Geophysics,
    // see http://rosettacode.org/wiki/Haversine_formula
    R: 6371000,

    // distance between two geographical points using spherical law of cosines ap
    distance: function (latlng1, latlng2) {···
    }
});
```

弄了一个Earth对象，继承了CRS的所有属性，新增了distance方法，用来计算两个经纬度之间的距离。这个我们过。

```
var EPSG3857 = extend({}, Earth, {
    code: 'EPSG:3857',
    projection: SphericalMercator,

    transformation: (function () {
        var scale = 0.5 / (Math.PI * SphericalMercator.R);
        return toTransformation(scale, 0.5, -scale, 0.5);
    }())
});
```

吼吼？ `EPSG3857` 的定义不就出来了？果然在继承的对象中添加了 `projection` 对象。并将 `SphericalMercator` 赋值给了它。自然而然，我们就找到了 `EPSG:3857` 坐标系的投影算法：

```
var SphericalMercator = {

    R: 6378137,
    MAX_LATITUDE: 85.0511287798,

    project: function (latlng) {
        var d = Math.PI / 180,
            max = this.MAX_LATITUDE,
            lat = Math.max(Math.min(max, latlng.lat), -max),
            sin = Math.sin(lat * d);

        return new Point(
            this.R * latlng.lng * d,
            this.R * Math.log((1 + sin) / (1 - sin)) / 2);
    },

    unproject: function (point) {
        var d = 180 / Math.PI;

        return new LatLng(
            (2 * Math.atan(Math.exp(point.y / this.R)) - (Math.PI / 2)) * d,
            point.x * d / this.R);
    },
```

到这里想必大家看明白了。为了验证我们的想法，马上去看了一下 `L.CRS` 提供的 `4326` 的算法。果然:结果如我们想象的一样:

```
var EPSG4326 = extend({}, Earth, {
    code: 'EPSG:4326',
    projection: LonLat,
    transformation: toTransformation(1 / 180, 1, -1 / 180, 0.5)
});
```

```
var LonLat = {
    project: function (latlng) {
        return new Point(latlng.lng, latlng.lat);
    },

    unproject: function (point) {
        return new LatLng(point.y, point.x);
    },

    bounds: new Bounds([-180, -90], [180, 90])
};
```

自己转自己，当然就是随意赋值给自己拉。

三、拓展坐标系

可惜。leaflet只提供了这2种坐标系。那如果我需要其他多种坐标系呢？给大家看看我们公司影像服务器要求支持的坐标系列表：

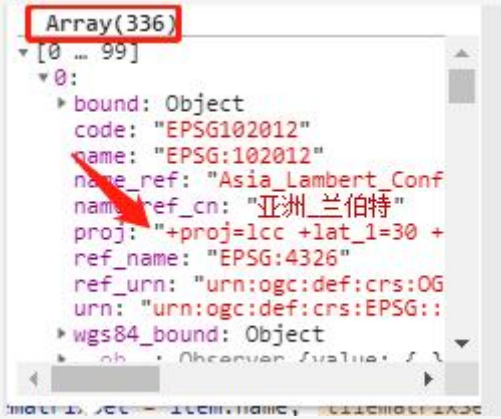
没错。有好几百个。。。实际存在的坐标系可能更多。那我们该如何拓展坐标系呢。看到这里，自然而然的我们就想到了gis神库: `proj4.js` 。 `proj4.js` 是一个开源的，专门用来进行坐标定义和坐标转换的工具。官档只有一个api:

```
proj4(fromProj, toProj, coord)
```

demo也很简单:

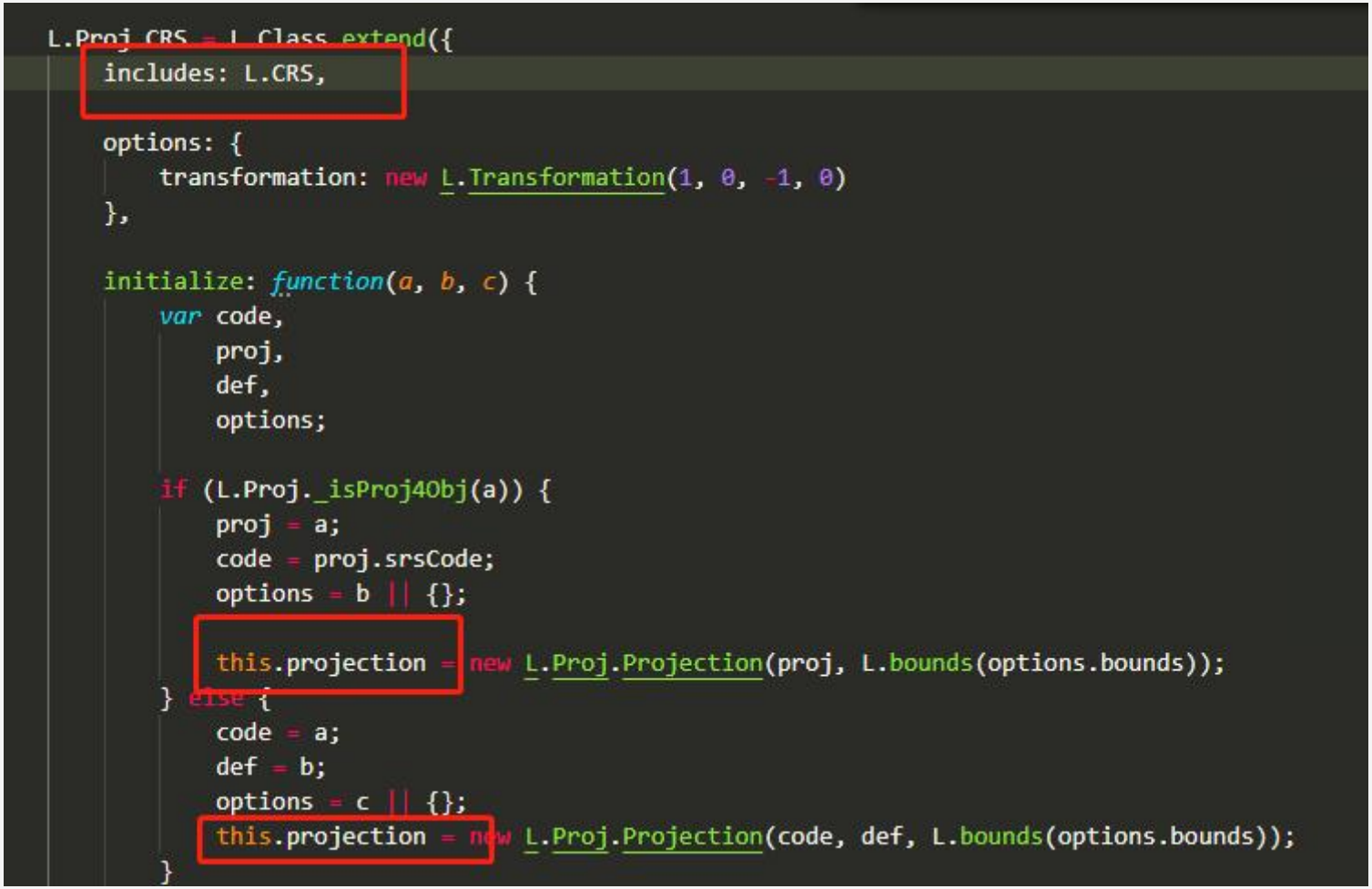
```
//2437自定义-->4326
var proj1 = '+proj=tmerc +lat_0=0 +lon_0=120 +k=1 +x_0=502000 +y_0=2000 +ellps=krass +towgs84=15.8,
var proj2 = '+proj=longlat +datum=WGS84 +no_defs';
var projPoint = proj4(proj1, proj2, [x, y]);
```

上面的那个奇怪的字符串就是该坐标系的算法定义。全国比较常用的，官档都有。你要说生僻的，那只能自己定义。关于里面的各个参数的意义，你让我说，我也没办法都给你说出来。反正在我看来，他就是一个字符串。巧了，服务器给我的坐标系数据里正好就有



现在算法有了。就是封装的过程。

我们的目的很明确，就是弄一个 `crs` 对象传递给map的初始化的crs属性。我这个 `crs` 对象得有一个 `projection` 子对象,这个子对象要有 `project` 和 `unproject` 函数，在这个函数里去调用 `proj4` 进行坐标转换。当然网上已经有不少做好了拓展封装。 `proj4leaflet.js` 就是一个。文末附上下载链接。简要摘出部分代码：



```
L.Proj.CRS = L.Class.extend({
    includes: L.CRS,

    options: {
        transformation: new L.Transformation(1, 0, -1, 0)
    },

    initialize: function(a, b, c) {
        var code,
            proj,
            def,
            options;

        if (L.Proj._isProj4Obj(a)) {
            proj = a;
            code = proj.srsCode;
            options = b || {};

            this.projection = new L.Proj.Projection(proj, L.bounds(options.bounds));
        } else {
            code = a;
            def = b;
            options = c || {};
            this.projection = new L.Proj.Projection(code, def, L.bounds(options.bounds));
        }
```

拓展自原生的 `L.CRS` ，初始化了 `projection` 对象。咋样？是不是和我说的一样？

在初始化函数里用 `proj4` 定义了一个Proj4对象。然后在project和unproject里调用函数进行转换。
调用方法：

```
const crs = new L.Proj.CRS(name,desc,{})

this.map = L.map("view-map", {
      zoomControl: false,
      attributionControl: false,
      crs: crs,
      minZoom: 1,
      maxZoom: 20,
  }).setView([38.00315, 114.28898], 4);
```

在第三个参数这里还可以配置原点和bounds。即该坐标系的原点和bbox。这些参数我们服务器都有，网上也都有。

结语

本人分析的比较浅，对于深层次的proj4的算法。表示真心看不懂。不得不说leaflet的设计还是相当灵活的。拓展性很强。网上很多插件都是基于leaflet做的拓展。如果有什么不明白的也欢迎留言~

proj4leaflet以及proj4下载地址