# 整体架构-jQuery3.2.1

# 创建jQuery对象----(L94)

# jQuery原型添加方法与属性

# 核心函数

# 选择器sizzle引擎

# 回调对象

# Deferred对象(异步队列)

# 数据缓存

# 队列操作

# 事件系统

# DOM操作

# 样式操作

# class操作

# 位置尺寸操作

# 属性操作

# 动画操作

## 概述

参考：http://www.cnblogs.com/aaronjs/p/4278660.html

### 动画参数

.animate( properties [, duration ] [, easing ] [, complete ] )  
.animate( properties, options )

* + - 1. 区别就与第二组数据的传递了，options是支持对象传参
      2. properties参数就是写一个CSS属性和值的对象，动画都是涉及变化的；
      3. 理论上来说有数值的属性都是可以变化的，width, height或者left可以执行动画，但是background-color不能，但是也不是绝对的，主要看数据的解析度，可以用插件支持
      4. 除了样式属性， 一些非样式的属性，如scrollTop 和 scrollLeft，以及自定义属性，也可应用于动画
      5. 除了定义数值，每个属性能使用'show', 'hide', 和 'toggle'。

### 举例说明

* + - 1. Html

<img id="book" alt="" width="100" height="123" style="background:red;opacity:1;position: relative; left: 500px;"/>

* + - 1. Script

$('#book').animate({  
 opacity: 0.25,  
 left: '50',  
 height: 'toggle'  
}, {  
 duration: 1000,  
 specialEasing: {  
 height: 'linear'  
 },  
 step: function (now, fx) {  
 console.log('step')  
 },  
 progress: function () {  
 console.log('progress')  
 },  
 complete: function () {  
 console.log('动画完成')  
 }  
})

* + - 1. **动画的参数都是最终值**
         1. 如上img元素的起始
         2. opacity是1，那么通过动画改成成0.25
         3. left是500，那么通过动画改成成50
         4. height为'toggle' 意味着如果是隐藏与显示的自动切换
         5. step：是针对opacity/left/height各自动画，每次改变通知三次
         6. progress 是把opacity/left/height看成一组了，每次改变只通知一次

### 动画每步运动的计算

* + - 1. 动画原理
         1. 动画原理很简单，依据定时器不断改变元素属性
         2. 常规动画思路：以width为例子，开始为100px，最终为200px，运行2000ms，那么算出每1ms的width宽，然后设置就可以实现动画效果了
      2. 计算方式1(以left为例)

起始位置：100px；

终止位置：200px；

运行时间：1000ms；

每秒需要移动距离：(200-100)/1000ms = 0.1px

* + - * 1. **问题：**

Js为单线程，setTimeout、setInterval定时是向任务队列添加执行代码，但是必须等到队列中已有的代码执行完毕，而且，可能出现动画每一步中又调用animate，又有动画，即可能出现setTimeout的嵌套，时间更不准

浏览器并也并不一定严格按照设置的时间（比如20ms）来添加下一帧，IE8及以下浏览器的精度为15.625ms，IE9+、Chrome精度为4ms，ff和safari约为10ms。

因此：使用这种方式每一步走的距离并不一致

* + - 1. 计算方式2

由于setInterval设置duration=2s时间是固定的，浏览器每秒运行的帧数，默认设置为13ms，即13ms会调用一次tick函数

首先，运动开始时获取当前时间startTime =new Date();那运行一次tick函数（并不一定完全是13ms）后，剩余时间为remaining = (startTime + duration – new Date())，剩余时间就是开始时间+定时器设定时间-当前毫秒数

计算剩余时间占总设定时间比率rate= remaining/duration

那么，这次运动tick函数，实际走的时间占全部时间比率为1-rate

根据这次tick函数应该挪动的终点位置，(endPostion-startPosition) \* (1-rate) +startPostion

* + - * 1. 此种方式动画逻辑与定时器的时间、调用次数无关，因此整个动画可以使用一个统一的setInverval定时器

### requestAnimationFrame与setInterval

编写动画循环的关键是要知道延迟时间多长合适。一方面，循环间隔必须足够短，这样才能让不同的动画效果显得平滑流畅；另一方面，循环间隔还要足够长，这样才能确保浏览器有能力渲染产生的变化

大多数电脑显示器的刷新频率是60Hz，大概相当于每秒钟重绘60次。大多数浏览器都会对重绘操作加以限制，不超过显示器的重绘频率，因为即使超过那个频率用户体验也不会有提升。因此，最平滑动画的最佳循环间隔是1000ms/60，约等于16.6ms

setTimeout和setInterval的问题是，它们都不精确。它们的内在运行机制决定了时间间隔参数实际上只是指定了把动画代码添加到浏览器UI线程队列中以等待执行的时间。如果队列前面已经加入了其他任务，那动画代码就要等前面的任务完成后再执行

　requestAnimationFrame采用系统时间间隔，保持最佳绘制效率，不会因为间隔时间过短，造成过度绘制，增加开销；也不会因为间隔时间太长，使用动画卡顿不流畅，让各种网页动画效果能够有一个统一的刷新机制，从而节省系统资源，提高系统性能，改善视觉效果

## Tween

### 概述

* + - 1. 主要是获取和设置当前属性值
      2. 为每个属性提供一个运动easing(缓动函数)

### 函数构造--------------------------------(L6654)

// 动画举例，div，从100px挪动到200px  
// <div id="book" style="width: 100px;height: 100px;background:red;position: relative; left: 100px;"></div>  
// $('#book').animate({  
// left: '200'  
// },1000,function(){  
// console.log('ha')  
// })  
// Tween构造方法与jQuery类似  
// 生成单个属性的运动对象，创建动画缓动对象  
function Tween( elem, options, prop, end, easing ) {  
 return new Tween.prototype.init( elem, options, prop, end, easing );  
}  
// jQuery.Tween = new Tween.prototype.init( elem, options, prop, end, easing );  
jQuery.Tween = Tween;

### Tween.prototype------------------------------------(L6659)

// 主要是返回或设置当前元素，针对某个属性具体值  
// cur返回left当前值  
// run设置元素当前位置left值  
Tween.prototype = {  
 constructor: Tween,  
 init: function( elem, options, prop, end, easing, unit ) {  
 this.elem = elem;  
 this.prop = prop;  
 // 默认easing为swing  
 this.easing = easing || jQuery.easing.\_default;  
 this.options = options;  
 this.start = this.now = this.cur(); // start与end为开始与结束的样式值100,200无单位  
 this.end = end;  
 // 除了cssNumber中指定的可以为数字的属性，其它默认单位为px  
 this.unit = unit || ( jQuery.cssNumber[ prop ] ? "" : "px" );  
 },  
 // 计算当前样式值  
 // 如prop有propHooks的get方法，则使用，否则使用默认的  
 cur: function() {  
 var hooks = Tween.propHooks[ this.prop ];  
  
 return hooks && hooks.get ?  
 hooks.get( this ) :  
 Tween.propHooks.\_default.get( this );  
 },  
 // 会在tick函数中，传入percent，percent即是实际运行一次所占duration的比率  
 // tick函数理解为每帧调用的函数，即13ms一次  
 // 此函数是根据，传入的时间占比情况，计算时间样式变化占比，比如left从100到200px，计算每次left挪到到多少  
 run: function( percent ) {  
 var eased,  
 hooks = Tween.propHooks[ this.prop ];  
  
 if ( this.options.duration ) {  
 // 时间过了百分之x，并不代表需要运动百分之x的距离，调用easing对应的函数  
 // 可以在jQuery.easing中扩展运动函数，默认"swing"缓冲  
 // 默认jQuery.easing只是用了percent一个参数，但可以扩展  
 this.pos = eased = jQuery.easing[ this.easing ](  
 percent, this.options.duration \* percent, 0, 1, this.options.duration  
 );  
 } else {  
 // duration为0，则percent一定为1  
 this.pos = eased = percent;  
 }  
 // this.now为本次tick调用后，left应该到达的位置  
 this.now = ( this.end - this.start ) \* eased + this.start;  
 // options对象可以指定step函数，每个tween调用一次，都会被执行  
 if ( this.options.step ) {  
 this.options.step.call( this.elem, this.now, this );  
 }  
 // 根据计算好的时间，设置实际变化的属性，比如left从100到200px，每次运行tick后，  
 // 设置left应该到什么位置，比如是101px  
 if ( hooks && hooks.set ) {  
 hooks.set( this );  
 } else {  
 Tween.propHooks.\_default.set( this );  
 }  
 return this;  
 }  
};

Tween.prototype.init.prototype = Tween.prototype;

### Tween.propHooks

* + - 1. Tween.propHooks.\_default--------------------------------------------(L6705)

// 提供默认的获取和设置当前元素样式值的方法  
// get即是进行条件判断后调用jQuery.css  
// set即是jQuery.style或在属性上绑定值的一些判断  
Tween.propHooks = {  
 \_default: {  
 get: function( tween ) {  
 var result;  
  
 // 非dom节点，属性有值而style上无值的dom节点，均返回属性值  
 if ( tween.elem.nodeType !== 1 ||  
 tween.elem[ tween.prop ] != null && tween.elem.style[ tween.prop ] == null ) {  
 return tween.elem[ tween.prop ];  
 }  
  
 // 第三个参数为空字符串给.css,因为内部parseFloat处理，  
 // 会将值解析，去除单位，如果解析不了，则直接返回字符串  
 // 如10px传入去解析，为10；如传入aaa，会返回aaa  
 result = jQuery.css( tween.elem, tween.prop, "" );  
  
 // "", null, undefined and "auto" 转换为 0.  
 return !result || result === "auto" ? 0 : result;  
 },  
 set: function( tween ) {  
  
 // 使用step hook做后补  
 // 如有cssHook，使用  
 // 如.style可以使用，则使用  
 // Use .style if available and use plain properties where available.  
 if ( jQuery.fx.step[ tween.prop ] ) {  
 jQuery.fx.step[ tween.prop ]( tween );  
 // elem为元素节点，并且 是cssProps里面的，或是cssHook里面的，使用style设置  
 // 凡是执行run的，一定执行了cur函数，cur函数使用jQuery.css获取样式值；  
 // jQuery.css里面finalPropName函数会将prop存在cssProps里面，说明elem.style[修正属性]一定存在，至少返回""  
 } else if ( tween.elem.nodeType === 1 &&  
 ( tween.elem.style[ jQuery.cssProps[ tween.prop ] ] != null ||  
 jQuery.cssHooks[ tween.prop ] ) ) {  
 jQuery.style( tween.elem, tween.prop, tween.now + tween.unit );  
 } else {  
 // 最后，则把值设置到属性上  
 tween.elem[ tween.prop ] = tween.now;  
 }  
 }  
 }  
};

* + - 1. Tween.propHooks.scrollTop, Tween.propHooks.scrollLeft---------------(L6744)

// Support: IE <=9 only  
// 在非dom节点上设置，会出bug  
Tween.propHooks.scrollTop = Tween.propHooks.scrollLeft = {  
 set: function( tween ) {  
 // 节点类型，并且有父节点（根元素也有父节点，为document）  
 if ( tween.elem.nodeType && tween.elem.parentNode ) {  
 tween.elem[ tween.prop ] = tween.now;  
 }  
 }  
};

### jQuery.easing缓动函数--------------------------(L6754)

/\*\*  
 \* 如果要扩展动画效果，则扩展这个属性jQuery.easing  
 \*/  
jQuery.easing = {  
 // 线性运动  
 linear: function( p ) {  
 return p;  
 },  
 // 缓冲  
 swing: function( p ) {  
 return 0.5 - Math.cos( p \* Math.PI ) / 2;  
 },  
 \_default: "swing"  
};

## animate函数总览------------------(L7258)

* + - 1. 对于animate参数传递方式，可以查看<http://api.jquery.com/animate/>，可以看animate都是如何传递参数，以及各参数含义
      2. 先判断参数props是否为空对象
      3. 再利用jQuery.speed对参数进行处理
      4. 利用Animation构建动画函数
      5. 根据参数是否传递queue参数，决定函数是直接运行，还是扔入queue队列中

animate: function( prop, speed, easing, callback ) {  
 var empty = jQuery.isEmptyObject( prop ),  
 // 对参数进行处理，并根据opt.queue是为true 或==null，对回调函数callback进行可队列操作的包装  
 optall = jQuery.speed( speed, easing, callback ),  
 // 真正执行动画的地方是Animation函数  
 doAnimation = function() {  
  
 // 在特征的副本上操作，保证每个特征效果不会被丢失  
 var anim = Animation( this, jQuery.extend( {}, prop ), optall );  
  
 // 空动画或完成需要立马解决  
 if ( empty || dataPriv.get( this, "finish" ) ) {  
 // 删除动画队列里的动画  
 anim.stop( true );  
 }  
 };  
 doAnimation.finish = doAnimation;  
 // 由于参数option可以传递queue:false或queue：'aa'形式，  
 // 如为fasle，则在每个元素直接运行doAnimation函数  
 // 如为字符串形式'aa',则创建aa的动画序列，需要调用.dequeue("queuename")才能开始动画  
 return empty || optall.queue === false ?  
 this.each( doAnimation ) :  
 this.queue( optall.queue, doAnimation );// 因为jquery.fn.queue里面对于key为fx的队列，调用了dequeue出列函数，故动画会自动运行队列第一个  
},

## jQuery.speed------------------------(L7205)

* + - 1. animate有多种传入参数的方式，如.animate( properties [, duration ] [, easing ] [, complete ] )或.animate( properties, options )，speed函数先对各种参数传递进行修正；
      2. 对duration参数可以不是数值，如可以是slow，fast进行处理
      3. 如参数未设置queue，则设置动画默认队列为fx
      4. 对设定的动画完成函数complete，进行包装，增加队列操作
      5. 返回处理好的参数对象
      6. 为何将参数修正放在jQuery对象上，而不是写为内部函数，主要是为了给编写jQuery插件的开发者提供构建新的animation的方法

/\*\*  
 \* 方法主要是：修正参数，时间、算法、重写回调函数  
 \* 如下，2种调用方式，此函数将参数进行处理，并且修正callback回调函数, 加上dequeue的功能  
 \* 方法返回：包含了修正后的动画执行时间, 动画算法, 回调函数的对象.  
 \* 这个方法暴露在jQuery上，是为了可以给jQuery插件开发者提供构建新的animation方法的  
 \* animate调用方式：.animate( properties [, duration ] [, easing ] [, complete ] )，.animate( properties, options )  
 \* properties为动画终止时样式，因此animate的第二个参数起传入speed  
 \* 调用方式：jQuery.speed( [duration ] [, easing ] [, complete ] )  
 \* jQuery.speed( settings )  
 \* @param speed 动画运行时间  
 \* @param easing 动画的缓动效果  
 \* @param fn 动画完成时的回调  
 \* @return {{complete: (\*|boolean), duration: \*, easing: (\*|boolean)}}  
 \*/  
jQuery.speed = function( speed, easing, fn ) {  
 // 修正参数，如speed是对象，则返回opt为对象,对应jQuery.speed( settings )  
 var opt = speed && typeof speed === "object" ? jQuery.extend( {}, speed ) : {  
 // 如speed不是对象，对应jQuery.speed( [duration ] [, easing ] [, complete ] )  
 complete: fn || !fn && easing || // complete使用规则 fn>easing>speed  
 jQuery.isFunction( speed ) && speed,  
 duration: speed,  
 // 如fn为undefined，easing为func，则easing：fasle  
 easing: fn && easing || easing && !jQuery.isFunction( easing ) && easing  
 };  
  
 // 如jQuery.fx.off为true，直接到终点状态  
 // 动画可以通过外部设置jQuery.fx.off为true停止，如设置了所有动画会直接到达终止状态  
 if ( jQuery.fx.off ) {  
 opt.duration = 0;  
  
 } else {  
 if ( typeof opt.duration !== "number" ) {  
 // 如duration不是数值，则查看是否为speeds中设定的slow，fast等标志运动时常的字符串  
 // 如是，则获取定义的时常，否则用\_default时常  
 if ( opt.duration in jQuery.fx.speeds ) {  
 opt.duration = jQuery.fx.speeds[ opt.duration ];  
  
 } else {  
 opt.duration = jQuery.fx.speeds.\_default;  
 }  
 }  
 }  
  
 // true/undefined/null -> 设为默认队列"fx"  
 // optall.queue指定为false时，不使用queue队列机制，  
 // doAnimation将立即调用Animation执行动画，保留了原有的同步机制。  
 if ( opt.queue == null || opt.queue === true ) {  
 opt.queue = "fx";  
 }  
  
 // 保留用户设定的完成函数  
 opt.old = opt.complete;  
 // 将用户定义的动画完成回调函数进行包装  
 // 即增加队列操作，目的是该函数可以dequeue队列，让队列中下个doAnimation开始执行  
 opt.complete = function() {  
 if ( jQuery.isFunction( opt.old ) ) {  
 opt.old.call( this );  
 }  
 // 如opt.queue为true，则在调用了complete的同时，进行出列操作  
 if ( opt.queue ) {  
 jQuery.dequeue( this, opt.queue );  
 }  
 };  
  
 return opt;  
};

## function Animation()函数总览

* + - 1. 代码主要分为如下几个部分

function Animation( elem, properties, options ) {  
 var result,  
 stopped,  
 index = 0,  
 length = Animation.prefilters.length,

// 1、初始化一个延时对象，这个延时对象用来处理动画队列。  
 deferred = jQuery.Deferred().always( function() {...} ),

// 2、生成一个每一个时间点（相邻两个时间点的事件间隔默认为13毫秒）上都会执行的函数tick。  
 tick = function() {....},

// 3、生成动画用的所有特征组成的对象animation。  
 animation = deferred.promise(...),  
 props = animation.props;

// 4、属性修正。  
 propFilter( props, animation.opts.specialEasing );

// 5、调用defaultPrefilter做适配处理。  
 for ( ; index < length; index++ ) {  
 result = Animation.prefilters[ index ].call( animation, elem, props, animation.opts );  
 if ( result ) {....}  
 }

// 6、对每个属性调用创建缓动对象。  
 jQuery.map( props, createTween, animation );

// 7、创建动画开始时调用函数  
 if ( jQuery.isFunction( animation.opts.start ) ) {}

// 8、创建options里面的progress，complete等方法  
 animation  
 .progress( animation.opts.progress )  
 .done( animation.opts.done, animation.opts.complete )  
 .fail( animation.opts.fail )  
 .always( animation.opts.always );

// 9、启动动画计时,定时执行tick  
 jQuery.fx.timer(...);  
  
 return animation;  
}

## function propFilter()----------------(L7003)

* + - 1. 将传入的属性名，转为驼峰标记法
      2. 由于animate参数options可以传入specialEasing，故此函数还对缓动函数easing进行优先级处理
      3. 将属性为margin，padding的转换为marinLeft，marginRight等4个属性值

/\*\*  
 \* 属性修正。实际是为多种参数传递进行优先级划分  
 \* 属性修正指的是对动画传入的结束样式进行修正  
 \* 属性变为小驼峰，把还会利用cssHooks.expand把margin、padding、borderWidth拆分成4个方向  
 \* @param props 结束时的属性，{left:'200px', width:'500px'}  
 \* @param specialEasing 如可以在options中配置不同属性使用不同的缓动函数  
 \* specialEasing: {  
 \* width: "linear",  
 \* height: "easeOutBounce"  
 \* }  
 \*/  
function propFilter( props, specialEasing ) {  
 var index, name, easing, value, hooks;  
  
 // camelCase, specialEasing and expand cssHook pass  
 for ( index in props ) {  
 name = jQuery.camelCase( index );  
 easing = specialEasing[ name ];  
 value = props[ index ];  
 // easing优先级：value[ 1 ] > options.specialEasing[ name ] > options.easing  
 if ( Array.isArray( value ) ) {  
 easing = value[ 1 ];  
 value = props[ index ] = value[ 0 ];  
 }  
 // 修改属性名，将属性名转换为驼峰标记法的  
 if ( index !== name ) {  
 props[ name ] = value;  
 delete props[ index ];  
 }  
  
 hooks = jQuery.cssHooks[ name ];  
 if ( hooks && "expand" in hooks ) {  
 // expand扩展，margin/padding/border扩展为四个方向名值对形式  
 // 如margin:10px 5px 2px 1px 转为marginTop:10px,marginRight:5px的对象  
 value = hooks.expand( value );  
 // 删除原有margin/padding/border属性  
 delete props[ name ];  
  
 // 如在props指定了marginTop，则认为通过hooks转换的marginTop优先级更低  
 // 故不使用$.extend合并对象  
 // Reusing 'index' because we have the correct "name"  
 for ( index in value ) {  
 if ( !( index in props ) ) {  
 props[ index ] = value[ index ];  
 specialEasing[ index ] = easing;  
 }  
 }  
 } else {  
 specialEasing[ name ] = easing;  
 }  
 }  
}

## function defaultPrefilter()---------------(L6832)

* + - 1. 首先，根据传入queue为false，设置动画完成前调用complete函数，并删除dataPriv上的动画队列
      2. 针对属性值中有show，hide，toggle的进行处理，根据当前元素与动画最终元素是否相同，如都是show，则跳过，不同则将值转为当前元素实际属性值
      3. 对于width与height动画，针对display：none，inline,inline-box等不同display属性，进行处理，
      4. 完成动画最终的show，hide效果

/\*\*  
 \* 对某些属性进行适配处理，如height/width的动画要求display和overflow为特定的值才能有效果；  
 \* 比如对show/hide动画需要对一大堆css特征值进行动画  
 \* 1、对属性值有show，hide，toggle的进行处理  
 \* 2、对width，height动画时，不同display进行处理  
 \* @param elem 当前匹配元素  
 \* @param props 动画结束位置  
 \* @param opts 是jQuery.speed获取动画相关参数的对象  
 \*/  
function defaultPrefilter( elem, props, opts ) {  
 var prop, value, toggle, hooks, oldfire, propTween, restoreDisplay, display,  
 isBox = "width" in props || "height" in props,  
 anim = this,  
 orig = {},  
 style = elem.style,  
 hidden = elem.nodeType && isHiddenWithinTree( elem ),//判断是否为display:none  
 dataShow = dataPriv.get( elem, "fxshow" );  
  
 // animate调用时，可以给options传入queue:false参数，或一个字符串  
 // ?????????不是很理解这段，应该是保证定动画完成前，调用complete函数，并删除dataPriv上的动画队列  
 if ( !opts.queue ) {  
 hooks = jQuery.\_queueHooks( elem, "fx" );  
 if ( hooks.unqueued == null ) {  
 hooks.unqueued = 0;  
 oldfire = hooks.empty.fire;// 保存\_queueHooks的empty方法,empty:dataPriv.remove( elem, [ type + "queue", key ] );  
 hooks.empty.fire = function() {  
 if ( !hooks.unqueued ) {  
 oldfire();  
 }  
 };  
 }  
 hooks.unqueued++;  
  
 anim.always( function() {  
  
 // 确保动画在完成前会调用complete函数  
 anim.always( function() {  
 hooks.unqueued--;  
 if ( !jQuery.queue( elem, "fx" ).length ) {  
 hooks.empty.fire();  
 }  
 } );  
 } );  
 }  
  
 // 检测是否有show，hide或toggle动画效果，如类似height:toggle，会将高度逐渐减小到没有  
 // 如元素本身为show，动画终止又是show，则跳过，否则将show，hide，toggle，转为当前元素属性值  
 // $('#book').animate({  
 // width :'1000px',  
 // height: 'toggle'  
 // },1000,function(){  
 // console.log('ha')  
 // })  
 for ( prop in props ) {  
 value = props[ prop ];  
 // 动画种判断是否有toggle，show或者hide  
 // 主要是进行elem，display与设置show，hide是否一致，如本身为show，又设置show；  
 // 将hide，这些字符串进行转换；如elem原始为show，设置为hide，则将height设为datashow或elem本身高度  
 // 如元素原来是show，又设show，则删除height  
 if ( rfxtypes.test( value ) ) {  
 // 删除属性值为toggle，show，hide的属性，如上面例子，则删除height  
 delete props[ prop ];  
 toggle = toggle || value === "toggle";  
 // hidden为elem当前display是否为none，value为动画设置的值  
 // 如elem本身为hide，又设置为hide，或show又设为show  
 if ( value === ( hidden ? "hide" : "show" ) ) {  
  
 // 如value为show，并且fxshow有数据，则阻止元素show  
 if ( value === "show" && dataShow && dataShow[ prop ] !== undefined ) {  
 hidden = true;  
  
 // 否则跳出循环  
 } else {  
 continue;  
 }  
 }  
 // 如无dataShow数据，则绑定当前elem的prop样式值，（带单位）  
 orig[ prop ] = dataShow && dataShow[ prop ] || jQuery.style( elem, prop );  
 }  
 }  
  
 // 如props是空，且orig为空，则返回，没有动画结束位置，不进行动画操作  
 // 要先判断是否有show，hide等value值，再判断props是否为空  
 // 因为，终点动画只有一个值{height:'show'}；height被删除  
 // props为空，但如果elem本身为hide，则会在orig添加一个height属性，属性值为dataShow.height或元素本身高度  
 propTween = !jQuery.isEmptyObject( props );  
 if ( !propTween && jQuery.isEmptyObject( orig ) ) {  
 return;  
 }  
  
 // 如对width或height进行动画，针对display：none，inline,inline-box等不同display属性，应该如何处理  
 if ( isBox && elem.nodeType === 1 ) {  
  
 // Support: IE <=9 - 11, Edge 12 - 13  
 // 记录3个overflow属性，因为ie不能从overflowX，overflowY推断出overflow值  
 opts.overflow = [ style.overflow, style.overflowX, style.overflowY ];  
  
 // 如dataPriv获取不到fxshow上的display值，则获取dataPriv上的display值  
 // 获取存储的display值，这个主要用于动画种有show，hide，toggle  
 restoreDisplay = dataShow && dataShow.display;  
 if ( restoreDisplay == null ) {  
 restoreDisplay = dataPriv.get( elem, "display" );//设元素需要恢复的display属性值  
 }  
 // 获取当前元素的display值  
 display = jQuery.css( elem, "display" );  
 if ( display === "none" ) {  
 if ( restoreDisplay ) {  
 display = restoreDisplay;  
 } else {  
  
 // 通过临时设置元素可见性，好在之后获取width或height值（如非0）  
 showHide( [ elem ], true );// 显示元素  
 // 对于display:none元素，如内联设置了，showHide会将样式设置为""，showHide再细说  
 restoreDisplay = elem.style.display || restoreDisplay; // 获取内联display值  
 display = jQuery.css( elem, "display" );  
 showHide( [ elem ] );  
 }  
 }  
  
 // inline元素，以inline-box使之产生动画  
 // 如display===inline或 inline-block但restoreDisplay不为null或undefined  
 if ( display === "inline" || display === "inline-block" && restoreDisplay != null ) {  
 if ( jQuery.css( elem, "float" ) === "none" ) {  
  
 // propTween：为删除了height，width是否还有其他属性需要进行动画  
 // 如有返回false  
 if ( !propTween ) {  
 anim.done( function() {  
 style.display = restoreDisplay;  
 } );  
 if ( restoreDisplay == null ) {  
 display = style.display;  
 restoreDisplay = display === "none" ? "" : display;  
 }  
 }  
 style.display = "inline-block";  
 }  
 }  
 }  
 // 重点2:对于height/width动画overflow都要设置为"hidden"，动画完成后恢复。这个有利于提高渲染速度。  
 // 如是对width或height设置动画，则会设置opts.overflow属性  
 if ( opts.overflow ) {  
 style.overflow = "hidden";  
 // 不管调用的是deferred.resolve()还是deferred.reject()，最后总是执行。都会设置style的overflow3个属性  
 anim.always( function() {  
 style.overflow = opts.overflow[ 0 ];  
 style.overflowX = opts.overflow[ 1 ];  
 style.overflowY = opts.overflow[ 2 ];  
 } );  
 }  
  
 // 实现show,hide动画，这个动画效果相当于是在动画done之后，show或hide  
 propTween = false;  
 for ( prop in orig ) {  
  
 // 通用的元素 show/hide引擎  
 if ( !propTween ) {  
 if ( dataShow ) {  
 if ( "hidden" in dataShow ) {  
 hidden = dataShow.hidden;  
 }  
 } else {  
 dataShow = dataPriv.access( elem, "fxshow", { display: restoreDisplay } );  
 }  
  
 // 如属性值为toggle，存储hidden/visible值  
 if ( toggle ) {  
 dataShow.hidden = !hidden;  
 }  
  
 // 元素动画前先显示隐藏元素  
 // 对于隐藏元素，需要先show，才能看到动画效果  
 if ( hidden ) {  
 showHide( [ elem ], true );  
 }  
  
 /\* eslint-disable no-loop-func \*/  
  
 anim.done( function() {  
  
 /\* eslint-enable no-loop-func \*/  
  
 // “隐藏”动画的最后一步实际上是隐藏元素。  
 if ( !hidden ) {  
 showHide( [ elem ] );  
 }  
 dataPriv.remove( elem, "fxshow" );  
 // 给元素设置动画最终效果  
 for ( prop in orig ) {  
 jQuery.style( elem, prop, orig[ prop ] );  
 }  
 } );  
 }  
  
 // 生成属性独立的缓动对象  
 propTween = createTween( hidden ? dataShow[ prop ] : 0, prop, anim );  
 if ( !( prop in dataShow ) ) {  
 dataShow[ prop ] = propTween.start;  
 if ( hidden ) {  
 propTween.end = propTween.start;  
 propTween.start = 0;  
 }  
 }  
 }  
}

### Prefilters------------------------------------------(L7194)

* + - 1. Prefilters是动画过滤器数组，默认数组只有一个一个defaultPrefilter函数
      2. 通过prefilter可以为这个数组增加和删除过滤函数

// prefilters队列，默认只有defaultPrefilter一个函数  
prefilters: [ defaultPrefilter ],  
// 为prefilters添加删除callback函数使用，prepend：true表示添加  
// 比如，某天有border的div不能改width值形成动画，可以使用prefilter添加具体逻辑的函数  
prefilter: function( callback, prepend ) {  
 if ( prepend ) {  
 Animation.prefilters.unshift( callback );  
 } else {  
 Animation.prefilters.push( callback );  
 }  
}

## function createTween()

### function createTween( value, prop, animation )------------------------(L6818)

* + - 1. 主要是根据Animation.tweeners创建tween对象，即循环调用Animation.tweeners里面的函数

/\*\*  
 \* 创建tween对象  
 \* Animation中的调用，jQuery.map( props, createTween, animation );  
 \* 其中props假设为{left:'200px'},animation为Animation里面封装的Deferred对象  
 \* @param value 200px，注意这个value值是动画结束时的位置  
 \* @param prop left  
 \* @param animation  
 \* @return {\*}  
 \*/  
function createTween( value, prop, animation ) {  
 var tween,  
 // Animation.tweeners默认只有一个名为\*函数，可以通过插件扩展Animation.tweeners  
 collection = ( Animation.tweeners[ prop ] || [] ).concat( Animation.tweeners[ "\*" ] ),  
 index = 0,  
 length = collection.length;  
 for ( ; index < length; index++ ) {  
 if ( ( tween = collection[ index ].call( animation, prop, value ) ) ) {  
  
 // // 有返回值，则返回，不再遍历  
 return tween;  
 }  
 }  
}

### jQuery.Animation. tweeners--------------------------(L7176)

* + - 1. 默认情况下，此数组，只有一个函数，key为\*;主要工作是调用Animation函数里面animation对象里面的createTween函数，利用adjustCss对+=等数组进行调整
      2. 可以通过tweener函数，扩展jQuery.Animation.tweeners数组
      3. 因为，Animation函数调用jQuery.map(props,createTween,animation)，返回的是tween对象；

tweeners: {  
 // 会调用Animation的animation对象的createTween，创建动画结束时tween对象，value值为动画结束时位置  
 // 可以是100，100px,+=100;这些处理都是在adjustCSS中处理的，把单位放在tween.unit  
 "\*": [ function( prop, value ) {  
 // this.creteTween，调用的Animation函数中animation对象的createTween函数  
 // 主要是利用jQuery.Tween创建缓动对象，并将缓动对象压入animation.tweens中  
 // 注意，虽然压入animation.tweens数组，由于tween对象指向相同，更改了tween，animations.tweens数组对象对象也会更改  
 // 所以下面调用ajustCss对+=等数值进行调整，animations.tweens也会调整  
 var tween = this.createTween( prop, value );  
 adjustCSS( tween.elem, prop, rcssNum.exec( value ), tween );  
 // 返回的tween值不带单位，+=等也计算出结果  
 return tween;  
 } ]  
},

### this.createTween-------------------------(L7097)

* + - 1. 构建Tween对象

// 此方法是生成动画算法与流程控制器，即动画每一步运动到哪里，主要源于Tween的cur与run函数  
// 这只是定义createTween，创建各个属性缓动对象的函数  
createTween: function( prop, end ) {  
 var tween = jQuery.Tween( elem, animation.opts, prop, end,  
 animation.opts.specialEasing[ prop ] || animation.opts.easing );  
 // 将创建的缓动对象（每个属性会创建一个）推入tweens堆栈  
 animation.tweens.push( tween );  
 return tween;  
},

## jQuery.fx

### jQuery.fx.timer-------------------(L7418)

* + - 1. 函数将tick函数，tick函数包含了elem，anim，queue信息，传入到timers数组中
      2. 调用start函数，开始定时器，运行动画

// 把动画的tick函数加入$.timers堆栈  
// 此函数为Animation调用的函数，timer为Animation中定义的tick函数  
jQuery.fx.timer = function( timer ) {  
 jQuery.timers.push( timer );  
 jQuery.fx.start();  
};

### jQuery.fx.start-------------------------------(L7424)

* + - 1. 调用start会先判断inProgress是否为true，如为true，表示有定时器正在工作，则等待，否则去执行计划schedule函数

jQuery.fx.start = function() {  
 if ( inProgress ) {  
 return;  
 }  
  
 inProgress = true;  
 schedule();  
};

### function schedule()-------------------------------------------(L6777)

* + - 1. 如没有动画在执行，则创建定时器，执行tick函数，利用了setTimeout模拟setTimeInterval

function schedule() {  
 // 递归调用的用setTimeout模拟setInterval函数  
 if ( inProgress ) {  
 if ( document.hidden === false && window.requestAnimationFrame ) {  
 // requestAnimationFrame :<=ie9不支持  
 // requestAnimationFrame采用系统时间间隔，保持最佳绘制效率，不会因为间隔时间过短，造成过度绘制，增加开销；  
 // 也不会因为间隔时间太长，使用动画卡顿不流畅，让各种网页动画效果能够有一个统一的刷新机制，  
 // 从而节省系统资源，提高系统性能，改善视觉效果  
 // cancelAnimationFrame方法用于取消定时器,与setTimeout区别是不需要传事件间隔  
 window.requestAnimationFrame( schedule );  
 } else {  
 window.setTimeout( schedule, jQuery.fx.interval );  
 }  
  
 jQuery.fx.tick();  
 }  
}

### jQuery.fx.tick-------------------------(L7396)

* + - 1. 主要是运行timers里面的tick的函数，并在动画运行完删除tick

// jQuery.fx.tick() 被定时器调用，遍历timers堆栈  
jQuery.fx.tick = function() {  
 var timer,  
 i = 0,  
 timers = jQuery.timers;  
  
 fxNow = jQuery.now();  
  
 for ( ; i < timers.length; i++ ) {  
 // 在timers中取出一个tick函数  
 timer = timers[ i ];  
  
 // 每个动画的tick函数（即此处timer）执行时返回remaining剩余时间，结束返回false  
 // timers[ i ] === timer 的验证是因为可能有hooks在tick函数中删除某个timer，即允许外部删除timer  
 if ( !timer() && timers[ i ] === timer ) {  
 // 调用了就删除timer  
 timers.splice( i--, 1 );  
 }  
 }  
 // 无动画了，即timers.length=0，停止定时器  
 if ( !timers.length ) {  
 jQuery.fx.stop();  
 }  
 fxNow = undefined;  
};

### jQuery.fx.stop-----------------------------(L7433)

* + - 1. 暂停动画则直接将inProgress设置为null

// 停止全局定时器timerId  
jQuery.fx.stop = function() {  
 inProgress = null;  
};

## function Animation()源码解析------------(L7040)

/\*\*  
 \* 动画核心函数，返回animation  
 \* 重点参考：http://www.cnblogs.com/chuaWeb/p/jQuery-1-9-1-animate-1.html  
 \* @param elem  
 \* @param properties  
 \* @param options  
 \* @return {\*}  
 \* @constructor  
 \*/  
function Animation( elem, properties, options ) {  
 var result,  
 stopped,  
 index = 0,  
 // 默认length为1，只有一个defaultPrefilter，属性过滤器，prefilters  
 length = Animation.prefilters.length,  
 // 初始化一个延时对象，这个延时对象用来处理动画队列。  
 deferred = jQuery.Deferred().always( function() {  
  
 // 为了构建:animate选择，在动画结束后，无论成功与否，都删除tick上的elem  
 // 这样在:animate就返回elem===fn.elem匹配不到，返回false  
 delete tick.elem;  
 } ),  
 // 生成一个每一个时间点（相邻两个时间点的事件间隔默认为13毫秒）上都会执行的函数tick，  
 // 这个tick函数会保存在jQuery.timers中，然后每次执行jQuery.fx.tick的时候会取出来执行。  
 tick = function() {  
 if ( stopped ) {  
 return false;  
 }  
 // 根据动画每步计算方法2，获得每一步运行时，占用总时间的比率  
 var currentTime = fxNow || createFxNow(),  
 remaining = Math.max( 0, animation.startTime + animation.duration - currentTime ),  
  
 // Support: Android 2.3 only  
 // android使用这样语法 `1 - ( 0.5 || 0 )`会导致崩溃 (#12497)  
 temp = remaining / animation.duration || 0,  
 percent = 1 - temp,  
 index = 0,  
 length = animation.tweens.length;  
  
 for ( ; index < length; index++ ) {  
 // 传入百分比，把元素设置到合适位置  
 animation.tweens[ index ].run( percent );  
 }  
 // tick函数每调用一次，options.progress就执行一次  
 // notify动作是在progress这个Callback里  
 deferred.notifyWith( elem, [ animation, percent, remaining ] );  
  
 // 如percent小于1，则表示动画未完成，返回剩余时间  
  
 if ( percent < 1 && length ) {  
 return remaining;  
 }  
  
 // ？？？？？？？？？？？  
 if ( !length ) {  
 deferred.notifyWith( elem, [ animation, 1, 0 ] );  
 }  
  
 // resovle这个动画，报告结果  
 // 触发成功状态，会调用complete  
 deferred.resolveWith( elem, [ animation ] );  
 return false;  
 },  
 // 把对象中属性和值copy到deferred.promise中得到animation（一个promise对象）  
 // 生成动画用的所有特征组成的对象animation  
 animation = deferred.promise( {  
 elem: elem,  
 props: jQuery.extend( {}, properties ),  
 opts: jQuery.extend( true, {  
 specialEasing: {},  
 easing: jQuery.easing.\_default  
 }, options ),  
 originalProperties: properties,  
 originalOptions: options,  
 startTime: fxNow || createFxNow(),  
 duration: options.duration,  
 tweens: [],  
 // 此方法是生成动画算法与流程控制器，即动画每一步运动到哪里，主要源于Tween的cur与run函数  
 // 这只是定义createTween，创建各个属性缓动对象的函数  
 createTween: function( prop, end ) {  
 var tween = jQuery.Tween( elem, animation.opts, prop, end,  
 animation.opts.specialEasing[ prop ] || animation.opts.easing );  
 // 将创建的缓动对象（每个属性会创建一个）推入tweens堆栈  
 animation.tweens.push( tween );  
 return tween;  
 },  
 // 用于外部来停止动画的函数  
 stop: function( gotoEnd ) {  
 var index = 0,  
  
 // 如gotoEnd为true，则想停止tweens数组全部动画，  
 length = gotoEnd ? animation.tweens.length : 0;  
 if ( stopped ) {  
 return this;  
 }  
 stopped = true;  
 for ( ; index < length; index++ ) {  
 // percent为1，表示，直接运动到动画结束状态  
 // length被设为0，不进行run  
 animation.tweens[ index ].run( 1 );  
 }  
  
 // gotoEnd为true，则resovle，调用complete与progress  
 if ( gotoEnd ) {  
 deferred.notifyWith( elem, [ animation, 1, 0 ] );  
 deferred.resolveWith( elem, [ animation, gotoEnd ] );  
 } else {  
 // 触发reject  
 deferred.rejectWith( elem, [ animation, gotoEnd ] );  
 }  
 return this;  
 }  
 } ),  
 // animation.props中保存的是用户传入的特征（动画最终目标）。  
 props = animation.props;  
 // 属性修正。即是对传入的动画终止属性进行修正。并为多种参数传递进行优先级划分  
 propFilter( props, animation.opts.specialEasing );  
 // 调用defaultPrefilter做适配处理:比如对height/width的动画要求display和overflow为特定的值才能有效果；  
 // 比如对show/hide动画需要对一大堆css特征值进行动画，并且在函数里就调用createTweens生成缓动动画。  
 for ( ; index < length; index++ ) {  
 // 默认只有一项defalutPrefilter，show/hide/toggle机制处理、inline元素处理。无返回值  
 // 这里指的是如果自己通过jQuery.tweener()进行了拓展hook，即插件使用者，会获得result结果  
 result = Animation.prefilters[ index ].call( animation, elem, props, animation.opts );  
 if ( result ) {  
 if ( jQuery.isFunction( result.stop ) ) {  
 jQuery.\_queueHooks( animation.elem, animation.opts.queue ).stop =  
 // result.stop.bind(result)  
 jQuery.proxy( result.stop, result );  
 }  
 // 如有自己定义的动画，则不再返回默认Animation动画  
 return result;  
 }  
 }  
 // 对每个属性调用创建缓动对象，如props：{width:'500px',left :'1000px'}  
 // 会把结果推到animation.tweens中  
 // jQuery.map有返回值，返回值是tween对象数组，每个对象的样式值是通过ajustCss处理之后的  
 jQuery.map( props, createTween, animation );  
 // options可以写一个start：function(){}定义动画开始前执行的函数  
 if ( jQuery.isFunction( animation.opts.start ) ) {  
 animation.opts.start.call( elem, animation );  
 }  
  
 // 链式返回animation，从这里也可以看出options还可以指定progress、done、complete、fail、always函数  
 animation  
 .progress( animation.opts.progress )  
 .done( animation.opts.done, animation.opts.complete )  
 .fail( animation.opts.fail )  
 .always( animation.opts.always );  
 // 启动动画计时,定时执行tick  
 jQuery.fx.timer(  
 // 为tick添加elem，anim，queue，3个属性，并放入全局interval中  
 jQuery.extend( tick, {  
 elem: elem,  
 anim: animation,  
 queue: animation.opts.queue  
 } )  
 );  
  
 return animation;  
}

## $().stop()----------------------------(L7277)

/\*\*  
 \* 停止匹配元素正在运行的动画  
 \* 调用方式：.stop( [queue ] [, clearQueue ] [, jumpToEnd ] )  
 \* 根据api要求.stop( [clearQueue ] [, jumpToEnd ] )都为boolean类型  
 \* .stop( [queue ] [, clearQueue ] [, jumpToEnd ] )为string，boolean，boolean类型  
 \* queue： 需要停止的动画队列  
 \* clearQueue：是否删除队列中的动画，默认为false;比如,  
 \* $('#book').animate({left :'1000px'},1000); $('#book').animate({top :'1000px'},1000);  
 \* 这是两个队列，在动画运行第一个时，调用stop(),第二个动画还会运行，但如调用stop(true)则两个动画都停止  
 \* jumpToEnd，是否立即完成当前动画，默认为false,注意是当前动画，true表示动画直接到最终状态，false则是停在调用stop时元素属性的状态值  
 \* @param type  
 \* @param clearQueue  
 \* @param gotoEnd  
 \* @return {\*}  
 \*/  
stop: function( type, clearQueue, gotoEnd ) {  
 var stopQueue = function( hooks ) {  
 var stop = hooks.stop;  
 delete hooks.stop;  
 stop( gotoEnd );  
 };  
 // 参数调整  
 if ( typeof type !== "string" ) {  
 gotoEnd = clearQueue;  
 clearQueue = type;  
 type = undefined;  
 }  
 // clearQueue为true，type为string，清除type动画队列的全部动画  
 // type不是string，则就是 undefined，不会是false  
 if ( clearQueue && type !== false ) {  
 this.queue( type || "fx", [] );  
 }  
  
 return this.each( function() {  
 var dequeue = true,  
 // type只有undefined和字符串两种可能  
 // 如无自定义动画队列，即type不为string，则type为undefined，返回type + "queueHooks"  
 // 否则对于默认动画队列，返回false  
 index = type != null && type + "queueHooks",  
 timers = jQuery.timers,  
 data = dataPriv.get( this );// 获取当前匹配元素elem的dataPriv，如有动画的话，上面会存储动画队列  
  
 if ( index ) {  
 if ( data[ index ] && data[ index ].stop ) {  
 stopQueue( data[ index ] );  
 }  
 } else {  
 for ( index in data ) {  
 if ( data[ index ] && data[ index ].stop && rrun.test( index ) ) {  
 stopQueue( data[ index ] );  
 }  
 }  
 }  
  
 for ( index = timers.length; index--; ) {  
 if ( timers[ index ].elem === this && // 匹配队列和元素,  
 ( type == null || timers[ index ].queue === type ) ) {  
 // anim.stop函数，如gotoEnd为true，true表示动画直接到最终状态，触发notifyWith，resolveWith  
 // 如gotoEnd为false，会触发deferred.rejectWith，即会在动画停止时调用fail函数  
 timers[ index ].anim.stop( gotoEnd );  
 dequeue = false;  
 timers.splice( index, 1 ); // 对于停止动画，则删除timers里面的当前tick函数  
 }  
 }  
  
 // gotoEnd为false，会执行出列动作，这样当前动画停止了，但后面的动画由于出列，则继续运行  
 // 比如说，直接调用jquery.fn.delay，然后调用stop，打算停止延迟，会到这里，因为delay并没有往timers增加ticks函数  
 if ( dequeue || !gotoEnd ) {  
 jQuery.dequeue( this, type );  
 }  
 } );  
},

## $().finish()--------------------------(L7329)

/\*\*  
 \* 停止当前动画，删除队列里全部动画，直接将动画设置为最终状态  
 \* 类似于.stop(true, true)，清除queue，并将当前动画跳到最终状态；  
 \* 但不同的是，finish，是将所有动画跳到最后  
 \* @param type  
 \* @return {\*}  
 \*/  
finish: function( type ) {  
 if ( type !== false ) {  
 type = type || "fx";  
 }  
 return this.each( function() {  
 var index,  
 data = dataPriv.get( this ),  
 queue = data[ type + "queue" ],  
 hooks = data[ type + "queueHooks" ],  
 timers = jQuery.timers,  
 length = queue ? queue.length : 0;  
  
 // 设置私有数据上动画结束标识  
 // 这样animate的，dataPriv.get( this, "finish" )就会为true，因此运行的动画会直接调用anim.stop  
 // doAnimation = function() {  
 // var anim = Animation( this, jQuery.extend( {}, prop ), optall );  
 // if ( empty || dataPriv.get( this, "finish" ) ) {  
 // anim.stop( true );  
 // }  
 // };  
 data.finish = true;  
  
 // 首先情况queue  
 jQuery.queue( this, type, [] );  
 // 如hooks有stop，则调用，如jQuery.fn.delay上的hooks.stop，清除定时器  
 // stop掉type对应的"非doAnimation"动画  
 if ( hooks && hooks.stop ) {  
 hooks.stop.call( this, true );  
 }  
  
 // 寻找任何正在运行的动画，直接到最终状态  
 for ( index = timers.length; index--; ) {  
 if ( timers[ index ].elem === this && timers[ index ].queue === type ) {  
 timers[ index ].anim.stop( true );  
 timers.splice( index, 1 );  
 }  
 }  
  
 // 将原队列的动画finish？？？？？？？？？？  
 for ( index = 0; index < length; index++ ) {  
 if ( queue[ index ] && queue[ index ].finish ) {  
 queue[ index ].finish.call( this );  
 }  
 }  
  
 // 删除finish标识  
 delete data.finish;  
 } );  
}

## jQuery.fn.delay---------------------(L7446)

// 基于Clint Helfers插件  
// https://web.archive.org/web/20100324014747/http://blindsignals.com/index.php/2009/07/jquery-delay/  
/\*\*  
 \* 设置一个计时器来延迟队列中后续项目的执行  
 \* .delay( duration [, queueName ] )  
 \* @param time 毫秒值，队列下一个函数执行需要延迟的毫秒值  
 \* @param type 队列名称，默认为fx  
 \* @return {\*}  
 \*/  
jQuery.fn.delay = function( time, type ) {  
 time = jQuery.fx ? jQuery.fx.speeds[ time ] || time : time; // 如时间为slow，fast，则用speeds数组值  
 type = type || "fx";  
 // 在当前队列增加一个定时器，  
 return this.queue( type, function( next, hooks ) {  
 var timeout = window.setTimeout( next, time );  
 hooks.stop = function() {  
 window.clearTimeout( timeout );  
 };  
 } );  
};

## $().fadeTo()------------------(L7250)

/\*\*  
 \* 调整匹配元素的透明度  
 \* .fadeTo( duration, opacity [, complete ] )，持续时间，opacity从0到1的值，complete：完成回调  
 \* .fadeTo( duration, opacity [, easing ] [, complete ] )，easing，缓动函数  
 \* 渐变，从0到to，不可见的也将可见  
 \* @param speed  
 \* @param to  
 \* @param easing  
 \* @param callback  
 \* @return {\*}  
 \*/  
fadeTo: function( speed, to, easing, callback ) {  
  
 // // 把所有隐藏元素的设为显示，并且透明度设为0（暂时看不见）  
 return this.filter( isHiddenWithinTree ).css( "opacity", 0 ).show()  
  
 // 回到this，所有元素opacity运动到to  
 .end().animate( { opacity: to }, speed, easing, callback );  
},

## slideDown,slideUp等动画快捷方式

### 对外接口--------------------------------(L7381)

* + - 1. 思路简单：就是构建好动画最终结果值，调用animate

// 构建某些动画的快捷方式  
jQuery.each( {  
 slideDown: genFx( "show" ),  
 slideUp: genFx( "hide" ),  
 slideToggle: genFx( "toggle" ),  
 fadeIn: { opacity: "show" },  
 fadeOut: { opacity: "hide" },  
 fadeToggle: { opacity: "toggle" }  
}, function( name, props ) {  
 jQuery.fn[ name ] = function( speed, easing, callback ) {  
 return this.animate( props, speed, easing, callback );  
 };  
} );

### function genFx( type, includeWidth )-----------------(L797)

// 为jQuery几种便捷的动画调用方式提供参数，构建slideDown，slideUp，slideToggle，无includeWidth参数  
// slide动画效果是高度逐渐减小或增大  
// 比如slideDown: genFx( "show" )  
function genFx( type, includeWidth ) {  
 var which,  
 i = 0,  
 attrs = { height: type };  
  
 // 如includeWidth为true，则利用cssExpand，将属性扩展为上下左右，步长为1  
 // 否则步长为2，忽略l左右  
 // 即将padding，margin样式终值设为type  
 includeWidth = includeWidth ? 1 : 0;  
 for ( ; i < 4; i += 2 - includeWidth ) {  
 which = cssExpand[ i ];  
 attrs[ "margin" + which ] = attrs[ "padding" + which ] = type;  
 }  
 // 如includeWidth为true，则width，opacity都设为type  
 if ( includeWidth ) {  
 attrs.opacity = attrs.width = type;  
 }  
  
 return attrs;  
}

## show,hide,toggle

### isHiddenWithinTree---------------------(L4494)

// 判断是否display：none；  
// 判断是否在dom上为了解决firefox的bug  
var isHiddenWithinTree = function( elem, el ) {  
  
 // isHiddenWithinTree可能会被jQuery.filter函数调用，那种情况下，elemetn是第二个参数  
 elem = el || elem;  
  
 // 内联样式胜过一切  
 return elem.style.display === "none" ||  
 elem.style.display === "" && // 如内联样式为"" ,判断是否计算样式display:none  
  
 // 否则检测计算属性  
 // Support: Firefox <=43 - 45  
 // 非DOM元素，会计算display为none，所以首先使用contains保证elem在dom上  
 jQuery.contains( elem.ownerDocument, elem ) &&  
  
 jQuery.css( elem, "display" ) === "none";  
 };

### getDefaultDisplay-------------------------(L4601)

/\*\*  
 \* 获取默认的display属性，在dom上增加一个空的元素，如elem是div，则append一个空的div  
 \* 然后利用jQuery.css( temp, "display" );获取当前元素的display属性，并储存在defaultDisplayMap中  
 \* 其实就是获取空的div，p等元素节点的display属性；因为元素可能是内联元素，可能是block元素等，show时候要保证正确  
 \* 不能给span，show之后变成display：block  
 \* @param elem  
 \* @return {\*}  
 \*/  
function getDefaultDisplay( elem ) {  
 var temp,  
 doc = elem.ownerDocument,  
 nodeName = elem.nodeName,  
 display = defaultDisplayMap[ nodeName ];  
  
 if ( display ) {  
 return display;  
 }  
  
 temp = doc.body.appendChild( doc.createElement( nodeName ) );  
 display = jQuery.css( temp, "display" );  
  
 temp.parentNode.removeChild( temp );  
  
 if ( display === "none" ) {  
 display = "block";  
 }  
 defaultDisplayMap[ nodeName ] = display;  
  
 return display;  
}

### showHide---------------------------------(L4626)

/\*\*  
 \* 显示与隐藏元素内部方法  
 \* @param elements 匹配的元素  
 \* @param show  
 \* @return {\*}  
 \*/  
function showHide( elements, show ) {  
 var display, elem,  
 values = [],  
 index = 0,  
 length = elements.length;  
  
 // 确定元素显示是否需要改变  
 for ( ; index < length; index++ ) {  
 elem = elements[ index ];  
 if ( !elem.style ) {  
 continue;  
 }  
  
 display = elem.style.display;// 获取匹配元素的display值  
 if ( show ) {  
  
 // 因为我们队级联隐藏元素强制可见性，因此需要在第一次循环时立即检测，除非display===none  
 // 如内联样式为none，且!values[ index ]为true，内联样式display会被修正为""  
 if ( display === "none" ) {  
 values[ index ] = dataPriv.get( elem, "display" ) || null;  
 if ( !values[ index ] ) {  
 elem.style.display = "";  
 }  
 }  
 // isHiddenWithinTree会判断计算样式，如元素内联设置none，style中又设置none，则会利用getDefaultDisplay获取值  
 // 如只有内联样式设置了display:none,则通过设置display=""为空字符串，就可以让元素显示  
 // 但如果外联样式还有display：none，这么设置不起作用  
 if ( elem.style.display === "" && isHiddenWithinTree( elem ) ) {  
 values[ index ] = getDefaultDisplay( elem );  
 }  
 } else {//show为false  
 if ( display !== "none" ) {  
 values[ index ] = "none";  
  
 // 覆盖之前存储的display值  
 dataPriv.set( elem, "display", display );  
 }  
 }  
 }  
  
 // 再循环一次，设置元素的style值  
 for ( index = 0; index < length; index++ ) {  
 if ( values[ index ] != null ) {  
 elements[ index ].style.display = values[ index ];  
 }  
 }  
  
 return elements;  
}

### show,hide,toggle对外接口

* + - 1. 此处创建的show,hide,toggle是show()这种方式调用的，此处创建的不是动画效果，仅是展现与隐藏dom元素

jQuery.fn.extend( {  
 // 此处定义的show，hide方法并不能传入参数，只是显示与隐藏元素  
 show: function() {  
 return showHide( this, true );  
 },  
 hide: function() {  
 return showHide( this );  
 },  
 toggle: function( state ) {  
 if ( typeof state === "boolean" ) {  
 return state ? this.show() : this.hide();  
 }  
  
 return this.each( function() {  
 if ( isHiddenWithinTree( this ) ) {  
 jQuery( this ).show();  
 } else {  
 jQuery( this ).hide();  
 }  
 } );  
 }  
} );

### Show,hide,toggle动画操作

* + - 1. 通过对传入参数的不同，选择调用不同的函数定义，类似函数重载

jQuery.each( [ "toggle", "show", "hide" ], function( i, name ) {  
 var cssFn = jQuery.fn[ name ]; // 获取$().show(),$().hide()，无参数函数定义  
 jQuery.fn[ name ] = function( speed, easing, callback ) {  
 // 为$().show增加实现动画操作，如speed为boolean，或==null，只是调用普通的显示与隐藏元素  
 // 否则形成动画操作  
 return speed == null || typeof speed === "boolean" ?  
 cssFn.apply( this, arguments ) :  
 this.animate( genFx( name, true ), speed, easing, callback );  
 };  
} );

## :hidden,:visible伪类匹配------------------(L9416)

* + - 1. Visible判断标准是判断当前width，height，getClientRects().length

// 构建:hidden，:visible的伪类  
jQuery.expr.pseudos.hidden = function( elem ) {  
 return !jQuery.expr.pseudos.visible( elem );  
};  
jQuery.expr.pseudos.visible = function( elem ) {  
 return !!( elem.offsetWidth || elem.offsetHeight || elem.getClientRects().length );  
};

# ajax

## 概述

### GET和POST最重要的区别

语义上，GET是获取指定URL上的资源，是读操作，重要的一点是不论对某个资源GET多少次，它的状态是不会改变的，在这个意义上，我们说GET是安全的（不是被密码学或者数据保护意义上的安全）。因为GET是安全的，所以GET返回的内容可以被浏览器，Cache服务器缓存起来

POST的语意是对指定资源“追加/添加”数据，所以是不安全的，每次提交的POST，参与的代码都会认为这个操作会修改操作对象资源的状态，于是，浏览器在你按下F5的时候会跳出确认框，缓存服务器不会缓存POST请求返回内容。

### jQuery在ajax做了什么工作

* + - 1. 现代浏览器都支持ajax，只是ie使用new window.ActiveXObject("Microsoft.XMLHTTP")，标准浏览器使用new window.XMLHttpRequest()）
      2. 但是，原生ajax不支持跨域，jQuery添加了这方面的处理，jQuery使用**所有的带有src属性的标签都不受同源策略的影响**这个特性，对于跨域使用script标签的src来请求路径
      3. jQuery对ajax处理，主要体现在对浏览器兼容，数据的处理及过滤，各种事件的封装上

### json与jsonp

* + - 1. JSON是一种数据交换格式，而JSONP是一种依靠开发人员的聪明才智创造出的一种非官方跨域数据交互协议
      2. Jsonp需要服务器配合
         1. 本地定义一个函数localHandler，参数为data

<script type="text/javascript">  
 var localHandler = function(data){  
 alert('我是本地函数，可以被跨域的remote.js文件调用，远程js带来的数据是：' + data.result);  
 };  
</script>  
<script type="text/javascript" src="http://remoteserver.com/remote.js"></script>

* + - * 1. 远程服务器返回的是这个函数的调用，将需要的数据利用参数的形式传递进来，实现跨越

localHandler({"result":"我是远程js带来的数据"});

### 什么是MIME Type

* + - 1. 浏览器是如何处理内容的。在浏览器中显示的内容有 HTML、有 XML、有 GIF、还有 Flash ……那么，浏览器是如何区分它们，决定什么内容用什么形式来显示呢？答案是 MIME Type，也就是该资源的媒体类型。
      2. 媒体类型通常是通过 HTTP 协议，由 Web 服务器告知浏览器的，更准确地说，是通过 Content-Type 来表示的，例如:Content-Type: text/HTML
      3. 表示内容是 text/HTML 类型，也就是超文本文件，是经过 ietf 组织协商，以 RFC 的形式作为建议的标准发布在网上的；
      4. 如果是某个客户端自己定义的格式，一般只能以 application/x- 开头。
      5. **Content-type与MIME Type**
         1. Content-type是HTTP报文里的首部字段；MIME Type为首部字段的值

## 表单序列化

### $(). serializeArray()------------------------------------(L8464)

/\*\*  
 \* 利用form.elements获取表单元素，然后删除需要提交的元素  
 \* 筛选出表单中需要提交的数据并以key/value键值对的对象数组格式返回  
 \* 如[{name: "a", value: "1"},{name: "a", value: "1"},{name: "a", value: "1"}]  
 \* 哪些会被序列化进来，根据的是W3C规则，必须要包含name属性，file的select，选择的数据不会被序列化  
 \* 如表单元素无value属性，序列化的value值为空  
 \* @return {\*}  
 \*/  
serializeArray: function() {  
 // 将form中的表单相关的元素取出来组成数组  
 return this.map( function() {  
  
 // form.elements 集合可返回包含表单中所有元素的数组。  
 // this如非表单form，则返回undefined  
 var elements = jQuery.prop( this, "elements" );  
 // 利用makeArray将类数组elements转换为普通数组  
 return elements ? jQuery.makeArray( elements ) : this;  
 } )  
 //过滤出为需要提交的表单元素（有name名称、非disabled元素、非提交按钮等元素、checkbox/radio的checked的元素）  
 .filter( function() {// 筛选出与指定表达式匹配的元素集合  
 var type = this.type;  
  
 // this.name 必须有name属性  
 // jQuery( this ).is( ":disabled" ) ：非disabled属性，使用is，可以判断 fieldset[disabled]  
 return this.name && !jQuery( this ).is( ":disabled" ) &&  
 // rsumbmittable，当前元素节点为可提交元素  
 // 不提交button，submit等类型元素  
 rsubmittable.test( this.nodeName ) && !rsubmitterTypes.test( type ) &&  
 ( this.checked || !rcheckableType.test( type ) );  
 } )  
 .map( function( i, elem ) {  
 // 获取当前元素的value值  
 var val = jQuery( this ).val();  
  
 if ( val == null ) {  
 return null;  
 }  
 // 如val是数组，调用map函数，转换为key-value形式，value  
 if ( Array.isArray( val ) ) {  
 return jQuery.map( val, function( val ) {  
 // 将\r?\n替换为\r\n  
 return { name: elem.name, value: val.replace( rCRLF, "\r\n" ) };  
 } );  
 }  
  
 return { name: elem.name, value: val.replace( rCRLF, "\r\n" ) };  
 // get()返回普通对象，否则为jQuery对象  
 } ).get();  
}

### function buildParams()------------------------(L8385)

/\*\*  
 \* 序列化类似{key:[[1,2],{key1:value1}]},  
 \* @param prefix 对象的key，即上述name   
 \* @param obj 对象的value，即上述的2  
 \* @param traditional $.param传入的第二个参数  
 \* @param add 用于拼接key= value 的函数  
 \*/  
function buildParams( prefix, obj, traditional, add ) {  
 var name;  
 // 如obj是数组  
 if ( Array.isArray( obj ) ) {  
  
 // 序列化数组每个元素  
 jQuery.each( obj, function( i, v ) {  
 // traditional 为true，或key是以[]结尾字符串  
 if ( traditional || rbracket.test( prefix ) ) {  
  
 // 将每个数组项视为标量，即v是具体数组，不是数组或obj  
 add( prefix, v );  
  
 } else {  
  
 // 如数组的每一项value值还是一个对象或数组，递归调用buildParams  
 // 如value是数组，key转换为key[],如value为obj，且!=nul，则转换为key[name]形式  
 buildParams(  
 prefix + "[" + ( typeof v === "object" && v != null ? i : "" ) + "]",  
 v,  
 traditional,  
 add  
 );  
 }  
 } );  
  
 } else if ( !traditional && jQuery.type( obj ) === "object" ) {  
  
 // 序列化对象  
 for ( name in obj ) {  
 buildParams( prefix + "[" + name + "]", obj[ name ], traditional, add );  
 }  
  
 } else {  
  
 // 序列化普通值  
 add( prefix, obj );  
 }  
}

### jQuery.param()------------------------------------(L8423)

/\*\*  
 \* 将key/value键值对的对象数组序列化为“key=value&key=value…”字符串  
 \* 主要进行的处理：将key/value成作为URI组件编码（保证key和value不会出现特殊符号，即保证了“=”分割的正确性）使用“&”链接  
 \* $.param()：返回的字符串是经过编码的，可以使用decodeURIComponent(jQuery.param())获得未编码的字符串，查看序列化对象结果  
 \* @param a 数组，普通搞对象，jQuery对象  
 \* @param traditional 根据buildParam可以看出，如traditional为true，则序列化对象，只一层，  
 \* 如myInfo = {intro:{html:5, css:3} }，不会将解析为intro[html]=5,intro[css]=3  
 \* @return {string}  
 \*/  
jQuery.param = function( a, traditional ) {  
 var prefix,  
 s = [],  
 // 将key，value转换为key = value形式（编码过的）  
 add = function( key, valueOrFunction ) {  
  
 // 如value是function，则使用其返回值  
 var value = jQuery.isFunction( valueOrFunction ) ?  
 valueOrFunction() :  
 valueOrFunction;  
 // 将key，value进行编码  
 s[ s.length ] = encodeURIComponent( key ) + "=" +  
 encodeURIComponent( value == null ? "" : value );  
 };  
  
 // 如a传入的是数组,假设他是一个form表单键值对数组  
 if ( Array.isArray( a ) || ( a.jquery && !jQuery.isPlainObject( a ) ) ) {  
  
 // //序列化表单元素  
 jQuery.each( a, function() {  
 add( this.name, this.value );  
 } );  
 // 如传入对象，则有多种情况，如a = {key : [[1,2,3],1,3]},多层级数组，{key:{key1:value1}}多层级对象等  
 // 将复杂情况交给buildParams解析  
 } else {  
  
 // 如a是普通对象，调用buildParams函数  
 for ( prefix in a ) {  
 buildParams( prefix, a[ prefix ], traditional, add );  
 }  
 }  
  
 // 返回最终的序列化字符串，用&拼接  
 return s.join( "&" );  
};

### serialize: function()-----------------------------------------(L8461)

// 将form表单元素，直接序列化为字符串key=value&key1=value1等形式  
serialize: function() {  
 return jQuery.param( this.serializeArray() );  
},

## jQuery.ajaxSetup

### function ajaxExtend( target, src )------------------------------(L8595)

* + - 1. 针对ajax问题修复的extend函数

// ajax对象的extend，为了fix#9887问题，是浅复制，不是深复制  
// #9887问题是仅仅使用jQuery.extend会造成内存浪费  
// ajaxExtend 是用来将 ajax 函数参数进行标准化的，看看哪些参数没有赋值，让它等于默认值  
function ajaxExtend( target, src ) {  
 var key, deep,  
 flatOptions = jQuery.ajaxSettings.flatOptions || {};  
  
 for ( key in src ) {  
 // 复制的对象key不是undefined  
 if ( src[ key ] !== undefined ) {  
 // 不想对flatOptions中的key进行深复制，因此，对于这些key，直接改target上的值为新的src中的值  
 ( flatOptions[ key ] ? target : ( deep || ( deep = {} ) ) )[ key ] = src[ key ];  
 }  
 }  
 // 对非flatOptions的key，进行深复制，复制到target上  
 if ( deep ) {  
 jQuery.extend( true, target, deep );  
 }  
  
 return target;  
}

### jQuery.ajaxSetup--------------------------------------------(L8849)

* + - 1. 提供一个修改ajax默认参数的方法

// 用于设置AJAX的全局默认设置。  
// 默认情况下，不推荐更改，因为有些jQuery差距可能是基于ajax默认配置的  
// 使用ajaxSetting和用户传入的setting配置ajax默认全局配置  
// 如target函数忽略，则将最终结果写入ajaxSettings中  
// ajaxSetup，只是将对象extend，并没有对属性值进行处理  
ajaxSetup: function( target, settings ) {  
 return settings ?  
  
 // 构建一个settings对象,分别将jQuery.ajaxSettings与settings，拷贝到target上  
 ajaxExtend( ajaxExtend( target, jQuery.ajaxSettings ), settings ) :  
  
 // 扩展 ajaxSettings 对象，即jQuery.ajaxSetup(obj)调用，用target扩展jQuery.ajaxSettings  
 ajaxExtend( jQuery.ajaxSettings, target );  
},

### Jsonp的默认参数添加-------------------(L9664)

// 增加jsonp默认settings  
// 为jsonpCallback增加一个默认的function  
jQuery.ajaxSetup( {  
 // jsonp请求的参数名，如{jsonp:'onJSONPLoad'},则'onJSONPLoad=?'会传递给服务器  
 jsonp: "callback",  
 jsonpCallback: function() {  
 // nonce当前时间  
 var callback = oldCallbacks.pop() || ( jQuery.expando + "\_" + ( nonce++ ) );  
 this[ callback ] = true;  
 return callback;  
 }  
} );

### Script请求的默认参数---------------------(L9597)

// 增加script的dataType的默认配置  
// 为jQuery.ajaxSettings扩展  
jQuery.ajaxSetup( {  
 accepts: {  
 script: "text/javascript, application/javascript, " +  
 "application/ecmascript, application/x-ecmascript"  
 },  
 contents: {  
 script: /\b(?:java|ecma)script\b/  
 },  
 converters: {  
 "text script": function( text ) {  
 jQuery.globalEval( text );  
 return text;  
 }  
 }  
} );

### jQuery.ajaxSettings-------------------------------(L8781)

* + - 1. ajax默认参数设置
      2. 里面的converters用于将返回的text类型数据转换为需要的类型

// ajax全局默认settings  
// http://api.jquery.com/jQuery.ajax/官方api翻译过来  
ajaxSettings: {  
 url: location.href, // 请求URL  
 // http请求方法，get，post，put等  
 type: "GET",  
 // location.protocol=window.location.protocol，本地协议  
 // 允许当前环境被认为是本地的（如文件系统），即使默认情况下，jQuery不能识别  
 // 如下协议会别人为是本地的，file，\*-扩展协议，widget；  
 isLocal: rlocalProtocol.test( location.protocol ),  
 // 这个请求是否触发全局ajax事件，默认为true，如设置为false，则阻止类似ajaxStart，ajaxStop 等事件触发  
 global: true,  
 // options中的数据会被处理为查询字符串，如想发送DOMDocument或其他不想被处理的数据，可以设置为fasle  
 processData: true,  
 // 默认所有ajax操作都是异步的，如需要同步，设置为false  
 // 跨域请求和dataType：'jsonp'不支持同步请求  
 // 注意同步请求，会暂停浏览器工作，等待数据返回  
 async: true,  
 // 如下contentType适合大部分情况  
 // 如设置contentType：false，则jQuery不会设置任何contentType  
 contentType: "application/x-www-form-urlencoded; charset=UTF-8",  
  
 /\*  
 timeout: 0, // 0表示没有请求超时，timeout计时从调用$.ajax开始，  
 data: null, // 发给服务器的数据  
 dataType: null, // 服务器返回数据类型  
 username: null, //访问有权限的的HTTP需要的username  
 password: null,//访问有权限的的HTTP需要的password  
 cache: null, //如为false，则默认浏览器不缓存页面，设置false，只有在HEAD与get请求中有效，工作方式是在get请求后增加时间戳  
 throws: false,  
 traditional: false,//如设置为true，参数序列化就调用的是jQuery.param(obj,true)，不是深层序列化对象  
 headers: {},// 可以更改默认的header的 X-Requested-With: XMLHttpRequest，headers里面值同样可以在beforeSend函数中复写  
 \*/  
 // accepts依赖于DataType  
 // 不同type类型的MINI类型，确定服务器返回的数据类型  
 accepts: {  
 "\*": allTypes,  
 text: "text/plain",  
 html: "text/html",  
 xml: "application/xml, text/xml",  
 json: "application/json, text/javascript"  
 },  
 // jQuery如何解析响应的key/正则表达式的对象  
 contents: {  
 xml: /\bxml\b/,  
 html: /\bhtml/,  
 json: /\bjson\b/ //\b 匹配一个字边界，即字与空格间的位置。代表着单词的开头或结尾  
 },  
  
 responseFields: {  
 xml: "responseXML",  
 text: "responseText",  
 json: "responseJSON"  
 },  
  
 // 数据转换器  
 // 浏览器的XMLHttpRequest对象对数据的响应只有 responseText与responseXML 二种  
 // jQuery支持不同格式的数据返回形式,如xml，json，html等  
 // 服务器的传输返回的只能是string类型的数据，但是用户如果通过jQuery的dataType定义了json的格式后，  
 // 会默认把数据转换成Object的形式返回  
 // converters存储的返回数据处理的句柄  
 converters: {  
  
 // 转换为text  
 "\* text": String,  
  
 // Text 转换为html  
 "text html": true,  
  
 // text转换为json  
 "text json": JSON.parse,  
  
 // text转换为xml  
 "text xml": jQuery.parseXML  
 },  
  
 // flatOptions标识的的不能进行深复制的属性  
 // 可以在这个对象加上不想深复制的自定义options  
 flatOptions: {  
 url: true,  
 context: true  
 }  
},

## jQuery. ajaxPrefilter

### prefilters变量声明--------------------(L8509)

* + - 1. addToPrefiltersOrTransports使用闭包的形式，将prefilters传入，并使用jQuery.ajaxPrefilter函数产生对不同请求类型的过滤

/\* Prefilters  
 \* 1) 对引入自定义类型非常有用(see ajax/jsonp.js for an example)  
 \* 2) 何时会被调用:  
 \* - 在请求transport之前  
 \* - 参数序列化之后(如s.processData为true，则s.data为string)  
 \* 3) key是dataType  
 \* 4) key可以是通用符\*  
 \* 5) 从具体的dataType开始执行，如需要，则会执行\*  
 \*/  
prefilters = {},

### addToPrefiltersOrTransports-----------------------(L8534)

/\*\*  
 \* jQuery.ajaxPrefilter and jQuery.ajaxTransport的基本构造器  
 \* jQuery.ajaxPrefilter调用方式：jQuery.ajaxPrefilter( [dataTypes ], handler )  
 \* dataTypes：dataTypes，可以是空格分隔的  
 \* handlers：之后ajax请求的  
 \* 使用闭包的原理，保存着structure的引用，负责将单一前置过滤和单一请求分发器分别放入prefilters和transports。  
 \* @param structure  
 \* @return {Function}  
 \*/  
function addToPrefiltersOrTransports( structure ) {  
 // 通过闭包访问structure  
 // 之所以能同时支持Prefilters和Transports，关键在于structure引用的时哪个对象  
 // dataTypeExpression是可选的，默认为\*  
 return function( dataTypeExpression, func ) {  
 // 修正参数，dataTypeExpression默认为\*  
 if ( typeof dataTypeExpression !== "string" ) {  
 func = dataTypeExpression;  
 dataTypeExpression = "\*";  
 }  
  
 var dataType,  
 i = 0,  
 // 将空格分隔的dataTypes转换为数组  
 dataTypes = dataTypeExpression.toLowerCase().match( rnothtmlwhite ) || [];  
 // 如func为函数  
 if ( jQuery.isFunction( func ) ) {  
  
 // 循环dataTypeExpression中的每一个dataType  
 while ( ( dataType = dataTypes[ i++ ] ) ) {  
  
 //如dataType是以+开始的，则将func放在structure数组开始  
 //shift用于移除数组中第一项并且返回该项，unshift用于在数组前面添加任意项并且返回数组长度!  
 if ( dataType[ 0 ] === "+" ) {  
 // 获取dataType数组1后面的全部  
 dataType = dataType.slice( 1 ) || "\*";  
 ( structure[ dataType ] = structure[ dataType ] || [] ).unshift( func );  
  
 // 否则添加到结尾  
 } else {  
 ( structure[ dataType ] = structure[ dataType ] || [] ).push( func );  
 }  
 }  
 }  
 };  
}

### jQuery. ajaxPrefilter ------------------------(L8862)

// 在发送请求或被$.ajax()处理之前，处理自定义Ajax的options，或修改已经存在的options  
// 前置过滤器,就是在特定的环境针对特定的情况做一些必要的兼容的处理  
// 通过闭包保持对prefilters的引用，将前置过滤器添加到prefilters  
ajaxPrefilter: addToPrefiltersOrTransports( prefilters ),

### prefilters[‘\*’]------------------------(L9590)

* + - 1. 对于prefilters的key为\*时，调用的如下函数
      2. 调用jQuery.ajaxPrefilter实际是往prefilters里面增加key，value值，对于相同的key，会形成一个function数组，可以看addToPrefiltersOrTransports

// 当没有明确dataType提供，避免scripts自动执行(See gh-2432)  
// 对于jQuery.ajaxPrefilter(func)调用方式，实际是往prefilters['\*']中添加函数  
jQuery.ajaxPrefilter( function( s ) {  
 if ( s.crossDomain ) {  
 s.contents.script = false;  
 }  
} );

### Prefilter[‘script’]------------------------------(L9614)

// 为sciprt增加cache和跨域的配置  
jQuery.ajaxPrefilter( "script", function( s ) {  
 // cache未设置则设置为false  
 if ( s.cache === undefined ) {  
 s.cache = false;  
 }  
 // 跨域未被禁用，强制类型为GET  
 if ( s.crossDomain ) {  
 s.type = "GET";  
 }  
} );

### Prefilter[‘json’]与prefilter[‘jsonp’]----------------------------(L9674)

/\*\*  
 \* 向前置过滤器对象中添加特定类型的过滤器  
 \* 1、对jsonp的url进行拼接  
 \* 2、对jsonpCallback进行处理  
 \* 参数:s:经过各种处理后的options，originalSettings：$.ajax()传入的参数，并未做修改，没有增加默认值，jqXHR：jQuery封装的xhr对象  
 \* jQuery.ajaxPrefilter作用：可以更改已经存在的options的逻辑，比如，可以为跨域时，将url更改一下  
 \* $.ajaxPrefilter(function( options ) {  
 \* if ( options.crossDomain ) {  
 \* options.url = "http://mydomain.net/proxy/" + encodeURIComponent( options.url );  
 \* options.crossDomain = false;  
 \* }  
 \* });  
 \* 如下表示：针对，json jsonp请求，需要做一些什么处理  
 \* 因此，经过jQuery.ajaxPrefilter( "json jsonp", func)，实际是prefilters变为，prefilters.json = func  
 \* prefilters.jsonp = func  
 \*  
 \*  
 \*/  
jQuery.ajaxPrefilter( "json jsonp", function( s, originalSettings, jqXHR ) {  
  
 var callbackName, overwritten, responseContainer,  
 // jsonp参数会覆盖请求url中的callback函数，故，如jsonp:'func',则'func=?'会被传给服务器  
 jsonProp = s.jsonp !== false && ( rjsonp.test( s.url ) ?  
 "url" :  
 // 如果json的url或data有jsonp的特征，会被当成jsonp处理  
 // 如s.data为string，contentType是  
 // 以application/x-www-form-urlencoded开始，s.data存在?? ，=?结尾，=?& 返回data  
 typeof s.data === "string" &&  
 ( s.contentType || "" )  
 .indexOf( "application/x-www-form-urlencoded" ) === 0 &&  
 rjsonp.test( s.data ) && "data"  
 );  
  
 if ( jsonProp || s.dataTypes[ 0 ] === "jsonp" ) {  
  
 // 指定jsonp请求的callback函数名  
 // 不配置这个jsonpCallback，jQuery会产生一个唯一的随机名字，这样更有利于管理请求，提供callback和error控制  
 // jsonpCallback如是函数，则callbackName为函数返回值  
 // 如为给callbackName复制，则用的是默认值，最开始时是：jQuery.expando + "\_" + ( new Date() )  
 callbackName = s.jsonpCallback = jQuery.isFunction( s.jsonpCallback ) ?  
 s.jsonpCallback() :  
 s.jsonpCallback; // jsonpCallback只是函数名  
  
 // 将callback插入到url或form数据  
 // 根据jsonP，拼接url  
 if ( jsonProp ) {  
 s[ jsonProp ] = s[ jsonProp ].replace( rjsonp, "$1" + callbackName );  
 } else if ( s.jsonp !== false ) {  
 s.url += ( rquery.test( s.url ) ? "&" : "?" ) + s.jsonp + "=" + callbackName;  
 }  
  
 // script脚本执行后，使用数据转换器，converters，将json取回  
 s.converters[ "script json" ] = function() {  
 if ( !responseContainer ) {  
 jQuery.error( callbackName + " was not called" );  
 }  
 return responseContainer[ 0 ];  
 };  
  
 // 强制dataType为json类型  
 // dataType为希望从服务器返回的数据类型  
 s.dataTypes[ 0 ] = "json";  
  
 // 在window上注册回调函数  
 overwritten = window[ callbackName ];// 如callbackName不是从oldCallbacks取出的，overwritten一定为undefined  
 window[ callbackName ] = function() {  
 responseContainer = arguments;  
 };  
 // 这个方法加入到jqXHR的always中，表示不管对服务器的请求是失败还是成功都应该回调!  
 jqXHR.always( function() {  
  
 // 如不存在前值overwrittern，删除 window[ callbackName ]  
 if ( overwritten === undefined ) {  
 jQuery( window ).removeProp( callbackName );  
  
 // 否则存在前值  
 } else {  
 window[ callbackName ] = overwritten;  
 }  
  
 // 因为jsonp默认jsonpCallback函数中，将this[ callback ] = true;故如jsonCallback使用默认的  
 // 会进入这个函数  
 if ( s[ callbackName ] ) {  
  
 s.jsonpCallback = originalSettings.jsonpCallback;  
  
 // 保存callback名，之后使用  
 oldCallbacks.push( callbackName );  
 }  
  
 // 如响应是一个函数，则调用它Call if it was a function and we have a response  
 if ( responseContainer && jQuery.isFunction( overwritten ) ) {  
 overwritten( responseContainer[ 0 ] );  
 }  
  
 responseContainer = overwritten = undefined;  
 } );  
  
 // 如dataType类型为json或jsonp，上述处理完毕后，会转给script的prefilter  
 return "script";  
 }  
} );

## jQuery.ajaxTransport

### transports变量声明--------------------(L8520)

/\* Transports 绑定  
 \* 1) key是dataType  
 \* 2) 通配符\*可以使用  
 \* 3) 从具体的dataType开始执行，如需要，则会执行\*  
 \*/  
transports = {},

### jQuery. ajaxTransport-----------------------(L8863)

// 请求分发器  
// 通过闭包保持对transports的引用，将请求分发器添加到transports  
// 创建一个处理Ajax数据实际传输的对象,  
// transport 需要提供send与abort两个方法，用于ajax中发送请求时使用  
// transport 被认为是prefilters与converters都不够用时的最后手段  
ajaxTransport: addToPrefiltersOrTransports( transports ),

### ajaxTransport[‘\*’]------------------------------(L9447)

* + - 1. ajax默认的分发器，jQuery.ajaxTransport会返回一个send，一个abort函数，相当于对xhr.send与xhr.abort的包装；主要是对元素xhr对象进行操作
      2. 默认分发器是指，所有请求都会走这个分发器（但也是有条件的，要求浏览器支持cors，或支持XMLHttpRequest并不跨越）
      3. 对于send函数主要工作是：
         1. 1、构建元素xhr对象，并为xhr.open传入参数
         2. 2、根据配置参数options，分别设置xhr的xhrFields，mimeType，headers
         3. 3、为onload，onerror，onabort绑定事件处理函数callback；这一步ajax做的是将3个事件绑定函数抽为一个；这个回调函数callback主要是根据不同情况，调用ajax主函数的done函数，并为其传递状态码，状态标识文字，响应头等；ajax主函数的done函数，实际是操作jqXHR对象的；
         4. 4、调用xhr.send发送请求
         5. 5、期间还处理ie9的bug

jQuery.ajaxTransport( function( options ) {  
 var callback, errorCallback;  
  
 // 如浏览器支持cors，或 支持XMLHttpRequest对象但不跨域  
 if ( support.cors || xhrSupported && !options.crossDomain ) {  
 return {  
 // complete是用于通知ajax请求完成的回调函数，即ajax主函数中的done  
 send: function( headers, complete ) {  
 var i,  
 xhr = options.xhr(); //获取xhr对象 new window.XMLHttpRequest();  
 // 初始化一个请求  
 // 调用open()方法并不会真正发送请求，而只是启动一个请求以备发送。  
 xhr.open(  
 options.type,  
 options.url,  
 options.async, // 是否异步操作，false为同步  
 options.username, //认证时用户名  
 options.password //认证时密码  
 );  
  
 // 应用自定义xhr  
 // xhrFields是fieldName-fieldValue的对象，用于设置给原生的XHR对象  
 if ( options.xhrFields ) {  
 for ( i in options.xhrFields ) {  
 xhr[ i ] = options.xhrFields[ i ];  
 }  
 }  
  
 // 通过options里面mimeType类型，可以复写XHR的mimeType类型  
 if ( options.mimeType && xhr.overrideMimeType ) {  
 xhr.overrideMimeType( options.mimeType );  
 }  
  
 // X-Requested-With 请求头用于在服务器端判断request来自Ajax请求还是传统请求。  
 // 对于跨域请求，由于一些问题，我们并不进  
 // (这个请求头可以在ajaxSetup中，或options.headers中设置  
 // 如是同域请求，如已经提供了此请求头，则不更改  
 // crossDomain：true为跨域请求，false为同域请求  
 if ( !options.crossDomain && !headers[ "X-Requested-With" ] ) {  
 headers[ "X-Requested-With" ] = "XMLHttpRequest";  
 }  
  
 // 设置请求头  
 // 元素xhr对象设置请求头  
 for ( i in headers ) {  
 xhr.setRequestHeader( i, headers[ i ] );  
 }  
  
 // 回调函数  
 // 用于构建，onload，onerror，onabort的回调函数  
 // onload:下载数据成功会触发;onerror:当发生网络异常的时候会触发;onabort:终止该请求  
 // 这3个事件对于每个响应应该只会触发一个，因此，在callback中需要先将全部的事件监听设置为null  
 callback = function( type ) {  
 return function() {  
 if ( callback ) {  
 callback = errorCallback = xhr.onload =  
 xhr.onerror = xhr.onabort = xhr.onreadystatechange = null;  
  
 if ( type === "abort" ) {  
 // 将终止该请求。当一个请求被终止，它的 readyState 属性将被置为0  
 xhr.abort();  
 } else if ( type === "error" ) {  
  
 // Support: IE <=9 only  
 // ie9手动abort会抛出异常，并且抛出异常，且状态码不为数字  
 if ( typeof xhr.status !== "number" ) {  
 complete( 0, "error" );  
 } else {  
 complete(  
  
 // File协议总是产生状态0; see #8605, #14207  
 xhr.status,  
 xhr.statusText  
 );  
 }  
 } else {  
 complete(  
 xhrSuccessStatus[ xhr.status ] || xhr.status,  
 xhr.statusText,  
  
 // Support: IE <=9 only  
 // IE9 对于xhr.responseType应为text的会返回binary(trac-11426)  
 // xhr.responseType默认值为text  
 ( xhr.responseType || "text" ) !== "text" ||  
 typeof xhr.responseText !== "string" ?  
 // xhr.response:返回response的body，根据responseType返回js对象，document，ArrayBuffer等  
 { binary: xhr.response } :  
 // 当responseType为text或者empty string类型时可以使用responseText属性  
 { text: xhr.responseText },  
 // 获取所有的HTTP响应头的数据  
 xhr.getAllResponseHeaders()  
 );  
 }  
 }  
 };  
 };  
  
 // 监听事件，下载数据成功会触发xhr.onload  
 xhr.onload = callback();  
 // 错误回调  
 errorCallback = xhr.onerror = callback( "error" );  
 // 为onabort事件绑定事件处理函数  
 // Support: IE 9 only  
 // 使用 onreadystatechange 代替onabort事件，用于处理为捕获的aborts  
 // onabort事件，在调用xhr.abort()后触发  
 if ( xhr.onabort !== undefined ) {  
 xhr.onabort = errorCallback;  
 } else {  
 // ie9的bug处理  
 xhr.onreadystatechange = function() {  
  
 // 检查状态码，  
 if ( xhr.readyState === 4 ) {  
  
 window.setTimeout( function() {  
 if ( callback ) {  
 errorCallback();  
 }  
 } );  
 }  
 };  
 }  
  
 // 构造abort的callback  
 callback = callback( "abort" );  
  
 try {  
  
 // 发送请求，xhr.send(data),data可以是ArrayBuffer，ArrayBufferView ，FormData 等，MDN有讲  
 xhr.send( options.hasContent && options.data || null );  
 } catch ( e ) {  
  
 // #14683: 只有在出错但没通知时，再次抛出  
 if ( callback ) {  
 throw e;  
 }  
 }  
 },  
  
 abort: function() {  
 if ( callback ) {  
 callback();  
 }  
 }  
 };  
 }  
} );

### ajaxTransport[‘script’]

* + - 1. 只是利用script标签实现跨域操作

/\*\*  
 \* 对于script标签，增加跨域支持  
 \*   
 \*/  
jQuery.ajaxTransport( "script", function( s ) {  
  
 // 对于srcipt，只是处理跨域请求  
 // s.crossDomain：true为跨域请求，false为同域请求  
 if ( s.crossDomain ) {  
 var script, callback;  
 return {  
 // s.scriptCharset：只用于script的transport中，设置script标签的charset  
 send: function( \_, complete ) {  
 // 构建script，增加charset，src属性，绑定load，error事件  
 script = jQuery( "<script>" ).prop( {  
 charset: s.scriptCharset,  
 src: s.url  
 } ).on(  
 "load error",  
 callback = function( evt ) {  
 script.remove();// 响应事件后，删除标签  
 callback = null;// 将事件处理函数设置为null  
 if ( evt ) {  
 complete( evt.type === "error" ? 404 : 200, evt.type );  
 }  
 }  
 );  
  
 // 使用原生DOM操作，避免使用domMainp出错  
 document.head.appendChild( script[ 0 ] );  
 },  
 abort: function() {  
 if ( callback ) {  
 callback();  
 }  
 }  
 };  
 }  
} );

## function ajaxHandleResponses--------------(L8614)

* + - 1. dataType类型的参数有很多，通过此函数解析出正确的类型参数；可能希望返回是json但实际服务器返回的是html，也要做处理
      2. 如传入\*，或dataType不传入，则默认为\*，使用服务器返回数据类型
      3. 判断服务器返回的数据类型是否能处理，如能处理会将类型增加到dataTypes数组中
      4. 判断dataTypes类型是否在response里面，如在，使用这个类型，不在的话，利用converters数组判断response的类型是否能解析

/\*\*  
 \* dataType类型的参数，可以是xml, json, script, or html 或者干脆为空  
 \* 处理ajax请求的响应，解析出正确的dataType类型  
 \* - 在content-type与传入参数的dataType中寻找正确的dataType  
 \* - 返回响应的响应内容  
 \* 参数：responses：表示的是ajax请求之后，实际服务器返回{ binary: xhr.response }或{ text: xhr.responseText }，通用ajaxTransport进行了封装  
 \*  
 \*/  
function ajaxHandleResponses( s, jqXHR, responses ) {  
  
 var ct, type, finalDataType, firstDataType,  
 contents = s.contents, // string-regular 对象，用来表示jQuery如何解析响应  
 dataTypes = s.dataTypes;// 传入参数中的希望服务器返回的数据类型  
  
 //如果用户传入的dataType是\*,或未传入，默认则是\*  
 // 就会经过这里的逻辑，最后通过mimeType或者返回的Content-Type来决定!  
 while ( dataTypes[ 0 ] === "\*" ) {  
 dataTypes.shift();  
 if ( ct === undefined ) {  
 // ct获取mimeType，content-Type为内容类型  
 ct = s.mimeType || jqXHR.getResponseHeader( "Content-Type" );  
 }  
 }  
  
 // 对于用户传入\*，则需要检查下服务器返回的类型，是否为已知类型  
 if ( ct ) {  
 for ( type in contents ) {  
 // 如是已知类型，则使用contents里面对应的正则返回结果  
 if ( contents[ type ] && contents[ type ].test( ct ) ) {  
 dataTypes.unshift( type );  
 break;  
 }  
 }  
 }  
  
 // 检查期望的从服务器获取的数据类型，是否在服务响应的responses中  
 if ( dataTypes[ 0 ] in responses ) {  
 finalDataType = dataTypes[ 0 ];  
 } else {  
  
 // 如不在，则尝试转换dataTypes  
 for ( type in responses ) {  
 // 根据s.converters进行转换，如配置中的options.dataType=json,则finalDataType为text  
 if ( !dataTypes[ 0 ] || s.converters[ type + " " + dataTypes[ 0 ] ] ) {  
 finalDataType = type;  
 break;  
 }  
 if ( !firstDataType ) {  
 firstDataType = type;  
 }  
 }  
  
 // 如转换不了，则尝试使用之前的  
 finalDataType = finalDataType || firstDataType;  
 }  
  
 // 如判断好了最终的datType，则将dataType添加到list中，并返回响应的响应内容  
 if ( finalDataType ) {  
 if ( finalDataType !== dataTypes[ 0 ] ) {  
 dataTypes.unshift( finalDataType );  
 }//如输入datTypes为json，经过转换可能是['text','json']  
 return responses[ finalDataType ];  
 }  
}

## function ajaxConvert-----------------------(L8673)

* + - 1. 主要是根据ajaxHandleResponses处理的dataTypes数组，进行数据转换，如需要json，则将响应数据转换为json等
      2. 对于jsonp，可能需要先转换为script，再转换为json，都是通过这个函数实现的

/\*\*  
 \* 响应类型的转换  
 \* 如输入datTypes为jsonp，经过转换可能是['text','script','json']  
 \* 类型转换器ajaxConvert根据请求时设置的数据类型，从jQuery. ajaxSettings.converters寻找对应的转换函数，  
 \* 假设有类型A数据和类型B数据，A要转换为B（A > B），首先在converters中查找能 A > B 对应的转换函数，  
 \* 如果没有找到，则采用曲线救国的路线，寻找类型C，使得类型A数据可以转换为类型C数据，类型C数据再转换为类型B数据，最终实现 A > B  
 \* @param s  
 \* @param response 响应主体  
 \* @param jqXHR  
 \* @param isSuccess 响应是否成功  
 \* @return {\*}  
 \*/  
function ajaxConvert( s, response, jqXHR, isSuccess ) {  
 var conv2, current, conv, tmp, prev,  
 converters = {},  
  
 // 拷贝一份dataTypes，因为转换时需要修改  
 dataTypes = s.dataTypes.slice();  
  
 // 构建converters对应关系,key为小写，  
 if ( dataTypes[ 1 ] ) {  
 for ( conv in s.converters ) {  
 converters[ conv.toLowerCase() ] = s.converters[ conv ];  
 }  
 }  
 // 取出第一个元素,一般为text  
 current = dataTypes.shift();  
  
 // 按dataTypes顺序转换  
 while ( current ) {  
 // 如current是responseFields的key，则在jqXHR对应的s.responseFields[ current ] 绑定响应实体  
 if ( s.responseFields[ current ] ) {  
 jqXHR[ s.responseFields[ current ] ] = response;  
 }  
  
 // 如有data过滤器，则使用  
 // dataFilter用来处理XMLHttpRequest的原始响应数据  
 if ( !prev && isSuccess && s.dataFilter ) {  
 response = s.dataFilter( response, s.dataType );  
 }  
  
 prev = current;  
 current = dataTypes.shift();// 第二次调用dataTypes.shift(),根据举例，此处为json  
  
 if ( current ) {  
  
 // 如current为\*,则还操作第一项  
 if ( current === "\*" ) {  
  
 current = prev;  
  
 // 转换响应，如prev不是\*且与current不相等  
 } else if ( prev !== "\*" && prev !== current ) {  
  
 // 寻找定义好的转换器，对于jsonp情况，则先去寻找"text script"转换器  
 conv = converters[ prev + " " + current ] || converters[ "\* " + current ];  
  
 // 比如输入options.datType为"html json",尝试将html转换为json  
 // 但是没有这个转换器  
 if ( !conv ) {  
 //converters["\* text"]=window.String  
 //converters["text html"]=true  
 //converters["text json"]=jQuery.parseJSON  
 //converters["text xml"]=jQuery.parseXML  
 for ( conv2 in converters ) {  
  
 // 将s.convsert中的类型转换表达式拆分,tmp[0] 源类型 tmp[1] 目标类型  
 tmp = conv2.split( " " );  
 // A:pre:html  
 // B:tmp[0] 基本为text  
 // C:current:json  
 // 通过循环converters，发现有C为json的converters,A直接转换不了C，可以先尝试B，C  
 if ( tmp[ 1 ] === current ) {// 存在B->c转换器  
  
 // 查看是否有A-B转换器，没有则用\*—->b代替  
 conv = converters[ prev + " " + tmp[ 0 ] ] ||  
 converters[ "\* " + tmp[ 0 ] ];  
 if ( conv ) {  
  
 // text html": true,意思是不需要转换,直接那来用  
 if ( conv === true ) {  
 conv = converters[ conv2 ];  
  
 // A->C的转换器不存在，那么先此时存在A-B转换器  
 // 将current设置为B，然后将dataTypes数组增加C，再循环转换  
 } else if ( converters[ conv2 ] !== true ) {  
 current = tmp[ 0 ];  
 dataTypes.unshift( tmp[ 1 ] );  
 }  
 break;  
 }  
 }  
 }  
 }  
  
 // 如conv不是true，则应用转换器  
 if ( conv !== true ) {  
  
 // 如throws为true  
 if ( conv && s.throws ) {  
 response = conv( response );  
 } else {  
 try {  
 response = conv( response );//比如text转JSON，调用的是JSON.parse，并不是所有文字都能解析，对于不能解析的会报错  
 } catch ( e ) {  
 // 如数据转换不了，则返回错误  
 return {  
 state: "parsererror",  
 // conv不存在，则报错，不能从a转换到b  
 error: conv ? e : "No conversion from " + prev + " to " + current  
 };  
 }  
 }  
 }  
 }  
 }  
 }  
  
 return { state: "success", data: response };  
}

## jQuery.ajax-------------------------(L8866)

### 函数概况

* + - 1. 由于此函数支持jQuery.ajax( url [, settings ] )与jQuery.ajax( [settings ] )两种方式调用，因此，函数开始先对参数进行处理

if ( typeof url === "object" ) {  
 options = url;  
 url = undefined;  
}  
  
options = options || {};

* + - 1. 定义一些ajax需要使用的变量，比如需要缓存请求头，响应头等
         1. 根据官网<https://api.jquery.com/jQuery.ajax/>的api可以看出，ajax能传入多个参数，但用户不能把所有参数都传入，因此需要一个默认参数与用户参数组合；使用到了jQuery.ajaxSetup；这个函数类似于extend函数

s = jQuery.ajaxSetup( {}, options ),

* + - * 1. 构建deferred，与callback，callback主要用于存储完成时调用的函数
        2. 封装构建jqXHR对象，这个对象是对xhr的一个封装，如getResponseHeader将响应头转换为key-value对象等；
      1. 为了避免外部改变ajax状态，故需要限制deferred对象

deferred.promise( jqXHR );

* + - 1. 对用户传入的参数如url，dataType，crossmain，data进行处理
      2. 针对不同的dataType进行预处理，比如希望服务器返回jsonp，需要先将jsonp转换为script，再转换为json；就是在特定的环境针对特定的情况做一些必要的兼容的处理

inspectPrefiltersOrTransports( prefilters, s, options, jqXHR );

* + - 1. 对用户传入的global，type，hascontent，cache，ifModified进行处理
      2. 设置jqXHR的请求头，
      3. 为deferred增加complete，error，success回调函数
      4. 获取请求分发，这个实际上是底层的xhr处理，如创建的通用分发器\*，实际就是创建xhr，操作xhr根据不同情况发生xhr请求；但对于script，又不能只是通用的发送xhr请求，可能script有跨域请求，需要在dom创建一个script标签，然后src设置为url再请求；

transport = inspectPrefiltersOrTransports( transports, s, options, jqXHR );

* + - 1. 对请求分发获得的结果进行处理，是abort还是send
      2. Ajax主函数中封装了done函数，这个函数会传入到请求分发器中，然后获取底层xhr对象获取到的标识状态符，标识状态文字，响应头等信息；然后done函数对这些信息进行处理

### 源码解析----------------------------(L8866)

// ajax主方法  
ajax: function( url, options ) {  
 // 1、为传递的参数做适配  
 // 如url是object，对参数进行调整，向前兼容1.5版本  
 // url可以包含在options中  
 if ( typeof url === "object" ) {  
 options = url;  
 url = undefined;  
 }  
  
 // 强制options是对象  
 options = options || {};  
 // 2、创建一些变量  
 var transport,  
  
 // 无anti-cache参数的url  
 // cacheURL其实就是不带时间戳的一部分url  
 cacheURL,  
  
 // 返回头  
 // 缓存xhr.getAllResponseHeaders()结果  
 responseHeadersString,  
 // 缓存xhr.getAllResponseHeaders()解析为key-value的obj结果  
 responseHeaders,  
  
 // 超时处理  
 timeoutTimer,  
  
 // Url清除变量  
 urlAnchor,  
  
 // 请求状态,send时为false，完成 时为true  
 completed,  
  
 // 是否全局事件需要派发  
 fireGlobals,  
  
 // 循环变量  
 i,  
  
 // url未缓存部分  
 // uncached其实就是带有时间戳的一部分url  
 uncached,  
  
 // 利用ajaxSetup构建最终options对象  
 // 返回的是：将jQuery.ajaxSettings与settings，拷贝到{}上  
 s = jQuery.ajaxSetup( {}, options ),  
  
 // options可以传入context，表示Ajax相关回调的context，默认是s  
 callbackContext = s.context || s,  
  
 // 对于一个DOm节点或jQuery对象集合，则全局事件的context为jQuery封装的对象，否则为jQuery.event对象  
 globalEventContext = s.context &&  
 ( callbackContext.nodeType || callbackContext.jquery ) ?  
 jQuery( callbackContext ) :  
 jQuery.event,  
  
 // deferred对象就是jQuery的回调函数解决方案，它解决了如何处理耗时操作的问题，对那些操作提供了更好的控制，以及统一的编程接口  
 deferred = jQuery.Deferred(),  
 // 所有的回调队列，不管任何时候增加的回调保证只触发一次  
 completeDeferred = jQuery.Callbacks( "once memory" ),  
  
 // 获取最终options的statusCode参数，默认是空对象  
 // 可以这么设置：当404时就alert  
 // $.ajax({  
 // statusCode: {  
 // 404: function() {  
 // alert( "page not found" );  
 // }  
 // }  
 // });  
 statusCode = s.statusCode || {},  
  
 // 请求头  
 requestHeaders = {}, // 请求头的value  
 requestHeadersNames = {},// 请求头的name  
  
 // 默认的abort信息  
 strAbort = "canceled",  
  
 // 是jquery-XMLHttpRequest简称，jqXHR 已经完全可以取代 XHR 对象了，函数都进行扩展了  
 // 在ajax方法中返回的是jqXHR一个包装对象，在这个对象里面混入了所有实现方法  
 jqXHR = {  
 // 0：未初始化。尚未调用open()方法。  
 // 1：启动。已经调用open()方法，但尚未调用send()方法。  
 // 2：发送。已经调用send()方法，但尚未接收到响应。  
 // 3：接收。已经接收到部分响应数据。  
 // 4：完成。已经接收到全部响应数据，而且已经可以在客户端使用了。  
 readyState: 0,  
 /\*\*  
 \* 当不传入dataType或传入\*时，在ajaxHandleResponse里面，会调用  
 \* jqXHR.getResponseHeader( "Content-Type" )  
 \* 实际是将请求头的多行文本，转换为key-value存储在responseHeaders  
 \* 并返回key对应的值  
 \* @param key  
 \* @return {null}  
 \*/  
 getResponseHeader: function( key ) {  
 var match;  
 if ( completed ) {  
 if ( !responseHeaders ) {  
 responseHeaders = {};  
 // 这个responseHeadersString，如对于默认分发器，会使用xhr.getAllResponseHeaders()  
 // 获取请求头  
 // m：允许多行匹配；g：全局匹配  
 // 因为请求头可能是多行文本，故使用m标记  
 // 实际一个括号匹配请求头的key，第二个匹配请求头的value  
 // rheaders = /^(.\*?):[ \t]\*([^\r\n]\*)$/mg,  
 while ( ( match = rheaders.exec( responseHeadersString ) ) ) {  
 responseHeaders[ match[ 1 ].toLowerCase() ] = match[ 2 ];  
 }  
 }  
 match = responseHeaders[ key.toLowerCase() ];  
 }  
 return match == null ? null : match;  
 },  
  
 // 获取响应头的纯字符串,xhr.getAllResponseHeaders()获得的结果  
 getAllResponseHeaders: function() {  
 return completed ? responseHeadersString : null;  
 },  
  
 // 缓存请求头  
 setRequestHeader: function( name, value ) {  
 if ( completed == null ) {  
 name = requestHeadersNames[ name.toLowerCase() ] =  
 requestHeadersNames[ name.toLowerCase() ] || name;  
 requestHeaders[ name ] = value;  
 }  
 return this;  
 },  
  
 // 复写响应的content-type头  
 overrideMimeType: function( type ) {  
 if ( completed == null ) {  
 s.mimeType = type;  
 }  
 return this;  
 },  
  
 /\*\*  
 \* 用于处理opitons中传入statusCode，如  
 \* statusCode:{  
 \* '404':function(){  
 \* console.log('404')  
 \* },  
 \* '200':function(){  
 \* console.log('200')  
 \* }  
 \* }  
 \* map即为后面的对象  
 \* @param map  
 \* @return {jqXHR}  
 \*/  
 statusCode: function( map ) {  
 var code;  
 if ( map ) {  
 if ( completed ) {  
  
 // 根据状态码执行回调  
 jqXHR.always( map[ jqXHR.status ] );  
 } else {  
  
 // 将新的回调储存在久的statusCode上  
 for ( code in map ) {  
 statusCode[ code ] = [ statusCode[ code ], map[ code ] ];  
 }  
 }  
 }  
 return this;  
 },  
  
 // 取消请求，statusText是取消请求时，显示的字  
 abort: function( statusText ) {  
 // 如未传入显示默认的stAbort  
 var finalText = statusText || strAbort;  
 if ( transport ) {  
 transport.abort( finalText );  
 }  
 done( 0, finalText );  
 return this;  
 }  
 };  
  
 // 返回deferred的只读版本 deferred.promise(),避免外部随意的触发deferred对象的回调函数，  
 // 很有可能在AJAX请求结束前就触发了回调函数（resolve），这就是与AJAX本身的逻辑相违背了。  
 // 为了避免回到地狱等，使用deferred对象  
 deferred.promise( jqXHR );  
  
 // 未无协议名的url增加，prefilters可能会使用个到，如url有参数，会保留  
 // 将以// 开头的url，转换为带协议名的  
 // 如//www.baidu.com，会被转换为http://www.baidu.com  
 s.url = ( ( url || s.url || location.href ) + "" )  
 .replace( rprotocol, location.protocol + "//" );  
  
 // type是method的别名  
 s.type = options.method || options.type || s.method || s.type;  
  
 // 提取dtaTypes列表  
 // dataTypes可以传入'text xml'，把text响应转换为XML  
 // "jsonp text xml",可以认为是请求为JSONP，收到为text，并将text转换为xml  
 s.dataTypes = ( s.dataType || "\*" ).toLowerCase().match( rnothtmlwhite ) || [ "" ];  
  
 // crossDomain设置为true，会强制发起一个跨域请求  
 // 同域下，服务器会返回另一个domain域  
 // 通过判断用户请求的protocol：host，与当前环境是否相同判断是否跨域  
 if ( s.crossDomain == null ) {  
 urlAnchor = document.createElement( "a" );  
  
 // Support: IE <=8 - 11, Edge 12 - 13  
 // 对于如下url，ie会抛出异常  
 // e.g. http://example.com:80x/  
 try {  
 urlAnchor.href = s.url;  
  
 // Support: IE <=8 - 11 only  
 // 如url地址是相对的，a标签的host属性设置不正确  
 urlAnchor.href = urlAnchor.href;  
 s.crossDomain = originAnchor.protocol + "//" + originAnchor.host !==  
 urlAnchor.protocol + "//" + urlAnchor.host;  
 } catch ( e ) {  
  
 // 如解析url报错，则认为url是跨域的  
 s.crossDomain = true;  
 }  
 }  
  
 // processData默认为true  
 // 默认情况下，通过data属性传递进来的数据，如果是一个对象(技术上讲，只要不是字符串)，  
 // 都会处理转化成一个查询字符串，以配合默认内容类型 "application/x-www-form-urlencoded"  
 // 如data不是字符串，则利用jQuery.param解析  
 // traditional是为了兼容jQuery<1.3.2行为的  
 if ( s.data && s.processData && typeof s.data !== "string" ) {  
 s.data = jQuery.param( s.data, s.traditional );  
 }  
  
 // 运行prefilters中的函数进行预处理  
 inspectPrefiltersOrTransports( prefilters, s, options, jqXHR );  
  
 // 如prefilter中使请求aborted，则直接return  
 // completed请求状态,send时为false，完成时为true  
 if ( completed ) {  
 return jqXHR;  
 }  
  
 // 如设置了global为true则触发全局事件  
 // 如jQuery.event无定义，则不要触发事件(AMD使用中会出现这个问题)(#15118)  
 // s.global默认值为true  
 fireGlobals = jQuery.event && s.global;  
  
 // 触发ajaxStart事件，jQuery.active开始为0，jQuery.active++先执行表达式再自增  
 // 如调用$.ajax({}),又调用一次$.ajax({})，第一次会触发ajaxStart  
 if ( fireGlobals && jQuery.active++ === 0 ) {  
 jQuery.event.trigger( "ajaxStart" );  
 }  
  
 // 大写请求类型  
 s.type = s.type.toUpperCase();  
  
 // hasContent，是否为get或head请求，如不是，为true  
 // rnoContent = /^(?:GET|HEAD)$/  
 s.hasContent = !rnoContent.test( s.type );  
  
 // Save the URL in case we're toying with the If-Modified-Since  
 // and/or If-None-Match header later on  
 // 删除hash，简化url操作  
 cacheURL = s.url.replace( rhash, "" );  
  
 // 对于get与Head请求,需要设置更多请求options参数  
 if ( !s.hasContent ) {  
  
 // 存储url后面的hash，之后可以放回  
 // uncached其实就是带有时间戳的一部分url  
 uncached = s.url.slice( cacheURL.length );  
  
 // 如data存在，则在url后拼接数据  
 // 这里的s.data已经被处理为查询字符串  
 if ( s.data ) {  
 // 如有data数据，则先将无时间戳的URL部分拼接上数据  
 cacheURL += ( rquery.test( cacheURL ) ? "&" : "?" ) + s.data;  
  
 // #9682: 删除s.data,以便之后不再使用  
 delete s.data;  
 }  
  
 // 如cache为false，在url增加时间戳，避免缓存  
 // false只能针对dataType为script和jsonp类型  
 if ( s.cache === false ) {  
 cacheURL = cacheURL.replace( rantiCache, "$1" );  
 // uncached为#后面的字符串  
 uncached = ( rquery.test( cacheURL ) ? "&" : "?" ) + "\_=" + ( nonce++ ) + uncached;  
 }  
  
 // 将有hash的字符串与无时间戳的url拼接在一起  
 // 会在hash前面添加数据，并增加一个时间戳  
 // 用户输入options.url为aaa.html/b/#/c/d  
 // 最终url应是类似于：aaa.html/b/?key=12&value=4&\_=123123123#/c/d  
 s.url = cacheURL + uncached;  
  
 // 如这是编码之后的form表单数据，将'%20'转换为 '+'  
 } else if ( s.data && s.processData &&  
 ( s.contentType || "" ).indexOf( "application/x-www-form-urlencoded" ) === 0 ) {  
 s.data = s.data.replace( r20, "+" );  
 }  
  
 // ifModified默认为false，表示仅在服务器数据改变时获取新数据。使用HTTP包Last-Modified头信息判断。  
 // Etag & If-None-Match:由服务器生成返回给前端,当你第一次发起HTTP请求时，服务器会返回一个Etag，  
 // 并在你第二次发起同一个请求时，客户端会同时发送一个If-None-Match，而它的值就是Etag的值  
 // 然后，服务器会比对这个客服端发送过来的Etag是否与服务器的相同，  
 // 如果相同，就将If-None-Match的值设为false，返回状态为304，客户端继续使用本地缓存  
 // 即ETag就是服务器生成的一个标记，用来标识返回值是否有变化。  
 // Last-Modified表示响应资源在服务器最后修改时间而已  
 // last-modified不足:  
 // Last-Modified标注的最后修改只能精确到秒级，如果某些文件在1秒钟以内，被修改多次的话，它将不能准确标注文件的修改时间；  
 // 如果某些文件会被定期生成，当有时内容并没有任何变化，但Last-Modified却改变了，导致文件没法使用缓存；  
 // 有可能存在服务器没有准确获取文件修改时间，或者与代理服务器时间不一致等情形。  
 if ( s.ifModified ) {  
 if ( jQuery.lastModified[ cacheURL ] ) {  
 jqXHR.setRequestHeader( "If-Modified-Since", jQuery.lastModified[ cacheURL ] );  
 }  
 // Etag是服务器自动生成或者由开发者生成的对应资源在服务器端的唯一标识符，能够更加准确的控制缓存。  
 if ( jQuery.etag[ cacheURL ] ) {  
 jqXHR.setRequestHeader( "If-None-Match", jQuery.etag[ cacheURL ] );  
 }  
 }  
  
 // 如果有数据传送，同时也指定了get,post方法，同时contentType也指定!  
 // 那么更加contentType添加一个头Content-Type！  
 if ( s.data && s.hasContent && s.contentType !== false || options.contentType ) {  
 jqXHR.setRequestHeader( "Content-Type", s.contentType );  
 }  
  
 // 根据dataType设置请求头  
 // Accept代表发送端（客户端）希望接受的数据类型  
 // Accept：text/xml:代表客户端希望接受的数据类型是xml类型  
 jqXHR.setRequestHeader(  
 "Accept",  
 // 如果dataTypes类型是accepts里面的，如是script,text,html,xml,json，则拼接, \*/\*; q=0.01  
 // 如dataTypes为\* ,则拼接""  
 // 如非设定在accepts里面的dataTypes，则直接使用accepts['\*']  
 s.dataTypes[ 0 ] && s.accepts[ s.dataTypes[ 0 ] ] ?  
 s.accepts[ s.dataTypes[ 0 ] ] +  
 // 如accepts.dataTypes返回不是\*,则在返回的accept后增加, \*/\*; q=0.01  
 ( s.dataTypes[ 0 ] !== "\*" ? ", " + allTypes + "; q=0.01" : "" ) :  
 s.accepts[ "\*" ]  
 );  
  
 // headers里面的key-value值，可以设置到setRequestHeader请求头中  
 // 主要是将请求头设置到jqXHR中  
 for ( i in s.headers ) {  
 jqXHR.setRequestHeader( i, s.headers[ i ] );  
 }  
  
 // 允许用户定义headers和mimeytpes，和提前取消请求  
 // beforeSend是请求前的一个函数，可以在请求之前修改jqXHR，设置用户自定义headers等  
 // 如返回false，会取消请求，传入beforeSend的参数是jqXHR和s  
 if ( s.beforeSend &&  
 ( s.beforeSend.call( callbackContext, jqXHR, s ) === false || completed ) ) {  
  
 // 取消请求，使用abort；  
 return jqXHR.abort();  
 }  
  
 // strAbort信息由canceled更改abort  
 strAbort = "abort";  
  
 // 为done，fail，complete添加回调函数  
 completeDeferred.add( s.complete );  
 jqXHR.done( s.success );  
 jqXHR.fail( s.error );  
  
 // 获取请求分发器，实际是用构建好的jqXHR，进行原生xhr调用，然后获得结果  
 transport = inspectPrefiltersOrTransports( transports, s, options, jqXHR );  
  
 // 如无请求分发，则自动abort  
 // 如有请求分发，则发送HTTP请求  
 if ( !transport ) {  
 done( -1, "No Transport" );  
 } else {  
 // 1：启动。已经调用open()方法，但尚未调用send()方法。  
 jqXHR.readyState = 1;  
  
 // 触发全局ajaxSend事件  
 if ( fireGlobals ) {  
 globalEventContext.trigger( "ajaxSend", [ jqXHR, s ] );  
 }  
  
 // 如ajaxSend内终止请求，停止  
 if ( completed ) {  
 return jqXHR;  
 }  
  
 // 超时  
 if ( s.async && s.timeout > 0 ) {  
 timeoutTimer = window.setTimeout( function() {  
 jqXHR.abort( "timeout" );  
 }, s.timeout );  
 }  
  
 try {  
 completed = false;  
 // 用于发送 HTTP 请求  
 // 此send为transport包装的  
 transport.send( requestHeaders, done );  
 } catch ( e ) {  
  
 // 完成后重新抛出异常  
 if ( completed ) {  
 throw e;  
 }  
  
 // 如抛出异常，就调用  
 done( -1, e );  
 }  
 }  
  
 // 任何事情完成后的回调函数  
 // status:readyState的标识状态  
 // nativeStatusText:状态标识文字  
 // responses:根据通用ajaxTransport的callback函数，{ binary: xhr.response }或{ text: xhr.responseText }  
 // headers:响应头  
 function done( status, nativeStatusText, responses, headers ) {  
 var isSuccess, success, error, response, modified,  
 statusText = nativeStatusText;  
  
 // 如请求完成，直接返回避免多次请求  
 if ( completed ) {  
 return;  
 }  
  
 completed = true;  
  
 // 如有超时处理的定时器，则清除  
 if ( timeoutTimer ) {  
 window.clearTimeout( timeoutTimer );  
 }  
  
 // 将transport设置为undefined，有利于垃圾回收机制早点将transport使用的内存收回  
 // 因为transport使用了闭包  
 // 无论jqXHR对象使用多长时间  
 transport = undefined;  
  
 // 缓存响应头  
 responseHeadersString = headers || "";  
  
 // 设置status状态，如大于0，则设置为已经接收到全部响应数据  
 jqXHR.readyState = status > 0 ? 4 : 0;  
  
 // 根据status判断是否响应成功  
 // 304:文档未改变  
 // 300以下都是表示成功，200，请求成功，201，请求已经被实现，202服务器接收请求  
 // https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#2\_1  
 isSuccess = status >= 200 && status < 300 || status === 304;  
  
 // 获取响应数据  
 if ( responses ) {  
 response = ajaxHandleResponses( s, jqXHR, responses );  
 }  
  
 // 利用ajaxConvert，将响应数据转换为dataType标识的类型  
 response = ajaxConvert( s, response, jqXHR, isSuccess );  
  
 // 如响应成功了，进行一些处理  
 if ( isSuccess ) {  
  
 //如果ifModified存在，那么就要设置If-Modified-Since和If-None-Match头!  
 if ( s.ifModified ) {  
 modified = jqXHR.getResponseHeader( "Last-Modified" );  
 if ( modified ) {  
 jQuery.lastModified[ cacheURL ] = modified;  
 }  
 modified = jqXHR.getResponseHeader( "etag" );  
 if ( modified ) {  
 jQuery.etag[ cacheURL ] = modified;  
 }  
 }  
  
 // 如无内容；服务器成功处理了请求，但不需要返回任何实体内容  
 // HEAD请求：只请求页面的首部，因此也是无内容的  
 if ( status === 204 || s.type === "HEAD" ) {  
 statusText = "nocontent";  
  
 // 内容无修改  
 } else if ( status === 304 ) {  
 statusText = "notmodified";  
  
 // 如有数据，则配置下如下参数，response是通过ajaxConvert转换后的  
 } else {  
 statusText = response.state;  
 success = response.data;  
 error = response.error;  
 isSuccess = !error;  
 }  
 } else {//如请求失败了；或传入参数status小于0  
  
 //提取传入的错误信息  
 error = statusText;  
 if ( status || !statusText ) { // 将status设置为0  
 statusText = "error";  
 if ( status < 0 ) {  
 status = 0;  
 }  
 }  
 }  
  
 // 给jqXHR对象设置status与statusText  
 jqXHR.status = status;  
 jqXHR.statusText = ( nativeStatusText || statusText ) + "";  
  
 // 成功resolve，失败reject  
 if ( isSuccess ) {  
 deferred.resolveWith( callbackContext, [ success, statusText, jqXHR ] );  
 } else {  
 deferred.rejectWith( callbackContext, [ jqXHR, statusText, error ] );  
 }  
  
 // 由于opions中可以传睿statusCode，如404：funcction，需要将对应的function处理一下  
 jqXHR.statusCode( statusCode );  
 statusCode = undefined;  
 // 如触发全局事件为true，则根据是否成功判断，触发的事件  
 if ( fireGlobals ) {  
 globalEventContext.trigger( isSuccess ? "ajaxSuccess" : "ajaxError",  
 [ jqXHR, s, isSuccess ? success : error ] );  
 }  
  
 // 调用fireWith触发所有的complete添加的回调  
 // 执行完成回调，是完成并不是成功，无论成功与否都会调用，只会执行一次  
 completeDeferred.fireWith( callbackContext, [ jqXHR, statusText ] );  
  
 if ( fireGlobals ) {  
 // 触发ajaxComplete事件  
 globalEventContext.trigger( "ajaxComplete", [ jqXHR, s ] );  
  
 //如果全局的ajax计数器已经是0了，那么就会触发ajaxStrop事件!  
 if ( !( --jQuery.active ) ) {  
 jQuery.event.trigger( "ajaxStop" );  
 }  
 }  
 }  
  
 return jqXHR;  
},

## 构建get,post,getScript,getJSON方法

### $.get,$.post方法-----------------------------(L9313)

* + - 1. Get,post方法调用的是ajax方法

// 构建get，post方法  
jQuery.each( [ "get", "post" ], function( i, method ) {  
 jQuery[ method ] = function( url, data, callback, type ) {  
  
 // 参数调整  
 if ( jQuery.isFunction( data ) ) {  
 type = type || callback;  
 callback = data;  
 data = undefined;  
 }  
  
 // 第一个参数可以是obj对象  
 return jQuery.ajax( jQuery.extend( {  
 url: url,  
 type: method,  
 dataType: type,  
 data: data,  
 success: callback  
 }, jQuery.isPlainObject( url ) && url ) );  
 };  
} );

### $.getJSON,$.getScript---------------------------------(L9304)

* + - 1. GetJSON，getScript方法调用的是get方法

getJSON: function( url, data, callback ) {  
 return jQuery.get( url, data, callback, "json" );  
},  
  
getScript: function( url, callback ) {  
 return jQuery.get( url, undefined, callback, "script" );  
}

## $().load()-----------------------------(L9818)

/\*\*  
 \* 从服务器获取数据，并将返回的html放在匹配元素内  
 \* @param url 请求url  
 \* @param params 请求发送带给服务器的数据  
 \* @param callback 请求完成的回调  
 \* @return {jQuery.fn}  
 \*/  
jQuery.fn.load = function( url, params, callback ) {  
 var selector, type, response,  
 self = this,  
 off = url.indexOf( " " );  
 // $( "#result" ).load( "ajax/test.html #container" );  
 // 解析这样的调用方式，在#result下插入text.html中#container的内容  
 if ( off > -1 ) {  
 // stripAndCollapse根据HTML spec协议处理空白，将value数组再拼接为'a b c'形式  
 selector = stripAndCollapse( url.slice( off ) );  
 url = url.slice( 0, off );  
 }  
  
 // 如params是函数  
 if ( jQuery.isFunction( params ) ) {  
  
 // 则认为params这个函数是回调  
 callback = params;  
 params = undefined;  
  
 // 如params是对象，则发送post请求  
 } else if ( params && typeof params === "object" ) {  
 type = "POST";  
 }  
  
 // 如有匹配元素  
 if ( self.length > 0 ) {  
 jQuery.ajax( {  
 url: url,  
  
 // 如type为undefined，则默认type请求  
 // 这里指明参数，不适用默认，因为用户可能会使用ajaxSetup复写这些参数  
 type: type || "GET",  
 dataType: "html",  
 data: params  
 } ).done( function( responseText ) {// 如有成功返回  
  
 // Save response for use in complete callback  
 response = arguments;  
  
 self.html( selector ?  
  
 // 如果指定了选择器，则在一个虚拟div中找到正确的元素。  
 // 排除脚本以避免IE被拒绝的错误  
 jQuery( "<div>" ).append( jQuery.parseHTML( responseText ) ).find( selector ) :  
  
 // 如无selector则使用全部的响应文本  
 responseText );  
  
 // 无论成功与否，都调用callback函数  
 } ).always( callback && function( jqXHR, status ) {  
 self.each( function() {  
 callback.apply( this, response || [ jqXHR.responseText, status, jqXHR ] );  
 } );  
 } );  
 }  
  
 return this;  
};

# 最后

## jQuery.holdReady----------------------(L10187)

/\*\*  
 \* 保存或释放ready事件  
 \* 方法用于延迟ready事件，动态script加载时的高级用法-。-  
 \* 要在ready事件之前调用，否则无效  
 \* @param hold  
 \*/  
jQuery.holdReady = function( hold ) {  
 if ( hold ) {  
 jQuery.readyWait++;  
 } else {  
 jQuery.ready( true );  
 }  
};

## jQuery的详尽使用

查看官网，<http://api.jquery.com/>；有时候并不知道jquery如何使用的，不知参数如何传递含义对理解源码难度加高，通过api，查看各参数函数，方便理解

### 文字参考

http://blog.csdn.net/vbdfforever/article/category/6104773