**构建自己的AngularJS，**

**第一部分：Scope和Digest**

原文链接：http://teropa.info/blog/2013/11/03/make-your-own-angular-part-1-scopes-and-digest.html

Angular是一个成熟和强大的JavaScript框架。它也是一个比较庞大的框架，在熟练掌握之前，需要领会它提出的很多新概念。很多Web 开发人员涌向Angular，有不少人面临同样的障碍。Digest到底是怎么做的？定义一个指令（directive）有哪些不同的方 法？Service和provider有什么区别？

Angular的文档挺不错的，第三方的资源也越来越丰富，想要学习一门新的技术，没什么方法比把它拆开研究其运作机制更好。

在这个系列的文章中，我将从无到有构建AngularJS的一个实现。随着逐步深入的讲解，读者将能对Angular的运作机制有一个深入的认识。

在第一部分中，读者将看到Angular的作用域是如何运作的，还有比如$eval, $digest, $apply这些东西怎么实现。Angular的脏检查逻辑看上去有些不可思议，但你将看到实际并非如此。

**基础知识**

在Github上，可以看到这个项目的全部源码。相比只复制一份下来，我更建议读者从无到有构建自己的实现，从不同角度探索代码的每个步骤。在本文 中，我嵌入了JSBin的一些代码，可以直接在文章中进行一些互动。（译者注：因为我在github上翻译，没法集成JSBin了，只能给链接……）

我们将使用Lo-Dash库来处理一些在数组和对象上的底层操作。Angular自身并未使用Lo-Dash，但是从我们的目的看，要尽量无视这些不太相关的比较底层的事情。当读者在代码中看到下划线（\_）的时候，那就是在调用Lo-Dash的功能。

我们还将使用console.assert函数做一些特别的测试。这个函数应该适用于所有现代JavaScript环境。

下面是使用Lo-Dash和assert函数的示例：

http://jsbin.com/UGOVUk/4/embed?js,console

**Scope对象**

Angular的Scope对象是POJO（简单的JavaScript对象），在它们上面，可以像对其他对象一样添加属性。Scope对象是用构造函数创建的，我们来写个最简单的版本：

function Scope() {

}

现在我们就可以使用new操作符来创建一个Scope对象了。我们也可以在它上面附加一些属性：

var aScope = new Scope();

aScope.firstName = 'Jane';

aScope.lastName = 'Smith';

这些属性没什么特别的。不需要调用特别的设置器（setter），赋值的时候也没什么限制。相反，在两个特别的函数：$watch和$digest之中发生了一些奇妙的事情。

**监控对象属性：$watch和$digest**

$watch和$digest是相辅相成的。两者一起，构成了Angular作用域的核心：数据变化的响应。

使用$watch，可以在Scope上添加一个监听器。当Scope上发生变更时，监听器会收到提示。给$watch指定如下两个函数，就可以创建一个监听器：

* 一个监控函数，用于指定所关注的那部分数据。
* 一个监听函数，用于在数据变更的时候接受提示。

作为一名Angular用户，一般来说，是监控一个表达式，而不是使用监控函数。监控表达式是一个字符串，比如说 “user.firstName”，通常在数据绑定，指令的属性，或者JavaScript代码中指定，它被Angular解析和编译成一个监控函数。在 这篇文章的后面部分我们会探讨这是如何做的。在这篇文章中，我们将使用稍微低级的方法直接提供监控功能。

为了实现$watch，我们需要存储注册过的所有监听器。我们在Scope构造函数上添加一个数组：

function Scope() {

this.$$watchers = [];

}

在Angular框架中，双美元符前缀$$表示这个变量被当作私有的来考虑，不应当在外部代码中调用。

现在我们可以定义$watch方法了。它接受两个函数作参数，把它们存储在$$watchers数组中。我们需要在每个Scope实例上存储这些函数，所以要把它放在Scope的原型上：

Scope.prototype.$watch = function(watchFn, listenerFn) {

var watcher = {

watchFn: watchFn,

listenerFn: listenerFn

};

this.$$watchers.push(watcher);

};

另外一面就是$digest函数。它执行了所有在作用域上注册过的监听器。我们来实现一个它的简化版，遍历所有监听器，调用它们的监听函数：

Scope.prototype.$digest = function() {

\_.forEach(this.$$watchers, function(watch) {

watch.listenerFn();

});

};

现在我们可以添加监听器，然后运行$digest了，这将会调用监听函数：

http://jsbin.com/oMaQoxa/2/embed?js,console

这些本身没什么大用，我们要的是能检测由监控函数指定的值是否确实变更了，然后调用监听函数。

**脏值检测**

如同上文所述，监听器的监听函数应当返回我们所关注的那部分数据的变化，通常，这部分数据就存在于作用域中。为了使得访问作用域更便利，在调用监控函数的时候，使用当前作用域作为实参。一个关注作用域上fiestName属性的监听器像这个样子：

function(scope) {

return scope.firstName;

}

这是监控函数的一般形式：从作用域获取一些值，然后返回。

$digest函数的作用是调用这个监控函数，并且比较它返回的值和上一次返回值的差异。如果不相同，监听器就是脏的，它的监听函数就应当被调用。

想要这么做，$digest需要记住每个监控函数上次返回的值。既然我们现在已经为每个监听器创建过一个对象，只要把上一次的值存在这上面就行了。下面是检测每个监控函数值变更的$digest新实现：

Scope.prototype.$digest = function() {

var self = this;

\_.forEach(this.$$watchers, function(watch) {

var newValue = watch.watchFn(self);

var oldValue = watch.last;

if (newValue !== oldValue) {

watch.listenerFn(newValue, oldValue, self);

}

watch.last = newValue;

});

};

对每个监听器，我们调用监控函数，把作用域自身当作实参传递进去，然后比较这个返回值和上次返回值，如果不同，就调用监听函数。方便起见，我们把新旧值和作用域都当作参数传递给监听函数。最终，我们把监听器的last属性设置成新返回的值，下一次可以用它来作比较。

有了这个实现之后，我们就可以看到在$digest调用的时候，监听函数是怎么执行的：

http://jsbin.com/OsITIZu/3/embed?js,console

我们已经实现了Angular作用域的本质：添加监听器，在digest里运行它们。

也已经可以看到几个关于Angular作用域的重要性能特性：

* 在作用域上添加数据本身并不会有性能折扣。如果没有监听器在监控某个属性，它在不在作用域上都无所谓。Angular并不会遍历作用域的属性，它遍历的是监听器。
* $digest里会调用每个监控函数，因此，最好关注监听器的数量，还有每个独立的监控函数或者表达式的性能。

**在Digest的时候获得提示**

如果你想在每次Angular的作用域被digest的时候得到通知，可以利用每次digest的时候挨个执行监听器这个事情，只要注册一个没有监听函数的监听器就可以了。

想要支持这个用例，我们需要在$watch里面检测是否监控函数被省略了，如果是这样，用个空函数来代替它：

Scope.prototype.$watch = function(watchFn, listenerFn) {

var watcher = {

watchFn: watchFn,

listenerFn: listenerFn || function() { }

};

this.$$watchers.push(watcher);

};

如果用了这个模式，需要记住，即使没有listenerFn，Angular也会寻找watchFn的返回值。如果返回了一个值，这个值会提交给脏检查。想要采用这个用法又想避免多余的事情，只要监控函数不返回任何值就行了。在这个例子里，监听器的值始终会是未定义的。

http://jsbin.com/OsITIZu/4/embed?js,console

这个实现的核心就这样，但是离最终的还是差太远了。比如说有个很典型的场景我们不能支持：监听函数自身也修改作用域上的属性。如果这个发生了，另外有个监听器在监控被修改的属性，有可能在同一个digest里面检测不到这个变动：

http://jsbin.com/eTIpUyE/2/embed?js,console

我们来修复这个问题。

**当数据脏的时候持续Digest**

我们需要改变一下digest，让它持续遍历所有监听器，直到监控的值停止变更。

首先，我们把现在的$digest函数改名为$$digestOnce，它把所有的监听器运行一次，返回一个布尔值，表示是否还有变更了：

Scope.prototype.$$digestOnce = function() {

var self = this;

var dirty;

\_.forEach(this.$$watchers, function(watch) {

var newValue = watch.watchFn(self);

var oldValue = watch.last;

if (newValue !== oldValue) {

watch.listenerFn(newValue, oldValue, self);

dirty = true;

}

watch.last = newValue;

});

return dirty;

};

然后，我们重新定义$digest，它作为一个“外层循环”来运行，当有变更发生的时候，调用$$digestOnce：

Scope.prototype.$digest = function() {

var dirty;

do {

dirty = this.$$digestOnce();

} while (dirty);

};

$digest现在至少运行每个监听器一次了。如果第一次运行完，有监控值发生变更了，标记为dirty，所有监听器再运行第二次。这会一直运行，直到所有监控的值都不再变化，整个局面稳定下来了。

Angular作用域里并不是真的有个函数叫做$$digestOnce，相反，digest循环都是包含在$digest里的。我们的目标更多是清晰度而不是性能，所以把内层循环封装成了一个函数。

下面是新的实现：

http://jsbin.com/Imoyosa/3/embed?js,console

我们现在可以对Angular的监听器有另外一个重要认识：它们可能在单次digest里面被执行多次。这也就是为什么人们经常说，监听器应当是幂 等的：一个监听器应当没有边界效应，或者边界效应只应当发生有限次。比如说，假设一个监控函数触发了一个Ajax请求，无法确定你的应用程序发了多少个请 求。

在我们现在的实现中，有一个明显的遗漏：如果两个监听器互相监控了对方产生的变更，会怎样？也就是说，如果状态始终不会稳定？这种情况展示在下面的代码里。在这个例子里，$digest调用被注释掉了，把注释去掉看看发生什么情况：

http://jsbin.com/eKEvOYa/3/embed?js,console

JSBin执行了一段时间之后就停止了（在我机器上大概跑了100,000次左右）。如果你在别的东西比如Node.js里跑，它会一直运行下去。

**放弃不稳定的digest**

我们要做的事情是，把digest的运行控制在一个可接受的迭代数量内。如果这么多次之后，作用域还在变更，就勇敢放手，宣布它永远不会稳定。在这个点上，我们会抛出一个异常，因为不管作用域的状态变成怎样，它都不太可能是用户想要的结果。

迭代的最大值称为TTL（short for Time To Live）。这个值默认是10，可能有点小（我们刚运行了这个digest 100,000次！），但是记住这是一个性能敏感的地方，因为digest经常被执行，而且每个digest运行了所有的监听器。用户也不太可能创建10 个以上链状的监听器。

事实上，Angular里面的TTL是可以调整的。我们将在后续文章讨论provider和依赖注入的时候再回顾这个话题。

我们继续，给外层digest循环添加一个循环计数器。如果达到了TTL，就抛出异常：

Scope.prototype.$digest = function() {

var ttl = 10;

var dirty;

do {

dirty = this.$$digestOnce();

if (dirty && !(ttl--)) {

throw "10 digest iterations reached";

}

} while (dirty);

};

下面是更新过的版本，可以让我们循环引用的监控例子抛出异常：

http://jsbin.com/uNapUWe/2/embed?js,console

这些应当已经把digest的事情说清楚了。

现在，我们把注意力转到如何检测变更上吧。

**基于值的脏检查**

我们曾经使用严格等于操作符(===)来比较新旧值，在绝大多数情况下，它是不错的，比如所有的基本类型（数字，字符串等等），也可以检测一个对象 或者数组是否变成新的了，但Angular还有一种办法来检测变更，用于检测当对象或者数组内部产生变更的时候。那就是：可以监控值的变更，而不是引用。

这类脏检查需要给$watch函数传入第三个布尔类型的可选参数当标志来开启。当这个标志为真的时候，基于值的检查开启。我们来重新定义$watch，接受这个参数，并且把它存在监听器里：

Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {

var watcher = {

watchFn: watchFn,

listenerFn: listenerFn,

valueEq: !!valueEq

};

this.$$watchers.push(watcher);

};

我们所做的一切是把这个标志加在监听器上，通过两次取反，强制转换为布尔类型。当用户调用$watch，没传入第三个参数的时候，valueEq会是未定义的，在监听器对象里就变成了false。

基于值的脏检查意味着如果新旧值是对象或者数组，我们必须遍历其中包含的所有内容。如果它们之间有任何差异，监听器就脏了。如果该值包含嵌套的对象或者数组，它也会递归地按值比较。

Angular内置了自己的相等检测函数，但是我们会用Lo-Dash提供的那个。让我们定义一个新函数，取两个值和一个布尔标志，并比较相应的值：

Scope.prototype.$$areEqual = function(newValue, oldValue, valueEq) {

if (valueEq) {

return \_.isEqual(newValue, oldValue);

} else {

return newValue === oldValue;

}

};

为了提示值的变化，我们也需要改变之前在每个监听器上存储旧值的方式。只存储当前值的引用是不够的，因为在这个值内部发生的变更也会生效到它的引用 上，$$areEqual方法比较同一个值的两个引用始终为真，监控不到变化，因此，我们需要建立当前值的深拷贝，并且把它们储存起来。

就像相等检测一样，Angular也内置了自己的深拷贝函数，但我们还是用Lo-Dash提供的。我们修改一下$digestOnce，在内部使用新的$$areEqual函数，如果需要的话，也复制最后一次的引用：

Scope.prototype.$$digestOnce = function() {

var self = this;

var dirty;

\_.forEach(this.$$watchers, function(watch) {

var newValue = watch.watchFn(self);

var oldValue = watch.last;

if (!self.$$areEqual(newValue, oldValue, watch.valueEq)) {

watch.listenerFn(newValue, oldValue, self);

dirty = true;

}

watch.last = (watch.valueEq ? \_.cloneDeep(newValue) : newValue);

});

return dirty;

};

现在我们可以看到两种脏检测方式的差异：

http://jsbin.com/ARiWENO/3/embed?js,console

相比检查引用，检查值的方式显然是一个更为复杂的操作。遍历嵌套的数据结构很花时间，保持深拷贝的数据也占用不少内存。这就是Angular默认不使用基于值的脏检测的原因，用户需要显式设置这个标记去打开它。

Angular也提供了第三种脏检测的方法：集合监控。就像基于值的检测，也能提示对象和数组中的变更。但不同于基于值的检测方式，它做的是一 个比较浅的检测，并不递归进入到深层去，所以它比基于值的检测效率更高。集合检测是通过“$watchCollection”函数来使用的，在这个系列的 后续部分，我们会来看看它是如何实现的。

在我们完成值的比对之前，还有些JavaScript怪事要处理一下。

**非数字（NaN）**

在JavaScript里，NaN（Not-a-Number）并不等于自身，这个听起来有点怪，但确实就这样。如果我们在脏检测函数里不显式处理NaN，一个值为NaN的监听器会一直是脏的。

对于基于值的脏检测来说，这个事情已经被Lo-Dash的isEqual函数处理掉了。对于基于引用的脏检测来说，我们需要自己处理。来修改一下$$areEqual函数的代码：

Scope.prototype.$$areEqual = function(newValue, oldValue, valueEq) {

if (valueEq) {

return \_.isEqual(newValue, oldValue);

} else {

return newValue === oldValue ||

(typeof newValue === 'number' && typeof oldValue === 'number' &&

isNaN(newValue) && isNaN(oldValue));

}

};

现在有NaN的监听器也正常了：

http://jsbin.com/ijINaRA/2/embed?js,console

基于值的检测实现好了，现在我们该把注意力集中到应用程序代码如何跟作用域打交道上了。

**$eval - 在作用域的上下文上执行代码**

在Angular中，有几种方式可以在作用域的上下文上执行代码，最简单的一种就是$eval。它使用一个函数作参数，所做的事情是立即执行这个传 入的函数，并且把作用域自身当作参数传递给它，返回的是这个函数的返回值。$eval也可以有第二个参数，它所做的仅仅是把这个参数传递给这个函数。

$eval的实现很简单：

Scope.prototype.$eval = function(expr, locals) {

return expr(this, locals);

};

$eval的使用一样很简单：

http://jsbin.com/UzaWUC/1/embed?js,console

那么，为什么要用这么一种明显很多余的方式去执行一个函数呢？有人觉得，有些代码是专门与作用域的内容打交道的，$eval让这一切更加明显。$scope也是构建$apply的一个部分，后面我们就来讲它。

然后，可能$eval最有意思的用法是当我们不传入函数，而是表达式。就像$watch一样，可以给$eval一个字符串表达式，它会把这个表达式编译，然后在作用域的上下文中执行。我们将在这个系列的后面部分实现这些。

**$apply - 集成外部代码与digest循环**

可能Scope上所有函数里最有名的就是$apply了。它被誉为将外部库集成到Angular的最标准的方式，这话有个不错的理由。

$apply使用函数作参数，它用$eval执行这个函数，然后通过$digest触发digest循环。下面是一个简单的实现：

Scope.prototype.$apply = function(expr) {

try {

return this.$eval(expr);

} finally {

this.$digest();

}

};

$digest的调用放置于finally块中，以确保即使函数抛出异常，也会执行digest。

关于$apply，大的想法是，我们可以执行一些与Angular无关的代码，这些代码也还是可以改变作用域上的东西，$apply可以保证作用域 上的监听器可以检测这些变更。当人们谈论使用$apply集成代码到“Angular生命周期”的时候，他们指的就是这个事情，也没什么比这更重要的了。

这里是$apply的实践：

http://jsbin.com/UzaWUC/2/embed?js,console

**延迟执行 - $evalAsync**

在JavaScript中，经常会有把一段代码“延迟”执行的情况 - 把它的执行延迟到当前的执行上下文结束之后的未来某个时间点。最常见的方式就是调用setTimeout()函数，传递一个0（或者非常小）作为延迟参数。

这种模式也适用于Angular程序，但更推荐的方式是使用$timeout服务，并且使用$apply把要延迟执行的函数集成到digest生命周期。

但在Angular中还有一种延迟代码的方式，那就是Scope上的$evalAsync函数。$evalAsync接受一个函数，把它列入计划， 在当前正持续的digest中或者下一次digest之前执行。举例来说，你可以在一个监听器的监听函数中延迟执行一些代码，即使它已经被延迟了，仍然会 在现有的digest遍历中被执行。

我们首先需要的是存储$evalAsync列入计划的任务，可以在Scope构造函数中初始化一个数组来做这事：

function Scope() {

this.$$watchers = [];

this.$$asyncQueue = [];

}

我们再来定义$evalAsync，它添加将在这个队列上执行的函数：

Scope.prototype.$evalAsync = function(expr) {

this.$$asyncQueue.push({scope: this, expression: expr});

};

我们显式在放入队列的对象上设置当前作用域，是为了使用作用域的继承，在这个系列的下一篇文章中，我们会讨论这个。

然后，我们在$digest中要做的第一件事就是从队列中取出每个东西，然后使用$eval来触发所有被延迟执行的函数：

Scope.prototype.$digest = function() {

var ttl = 10;

var dirty;

do {

while (this.$$asyncQueue.length) {

var asyncTask = this.$$asyncQueue.shift();

this.$eval(asyncTask.expression);

}

dirty = this.$$digestOnce();

if (dirty && !(ttl--)) {

throw "10 digest iterations reached";

}

} while (dirty);

};

这个实现保证了：如果当作用域还是脏的，就想把一个函数延迟执行，那这个函数会在稍后执行，但还处于同一个digest中。

下面是关于如何使用$evalAsync的一个示例：

http://jsbin.com/ilepOwI/1/embed?js,console

**作用域阶段**

$evalAsync做的另外一件事情是：如果现在没有其他的$digest在运行的话，把给定的$digest延迟执行。这意味着，无论什么时候调用$evalAsync，可以确定要延迟执行的这个函数会“很快”被执行，而不是等到其他什么东西来触发一次digest。

需要有一种机制让$evalAsync来检测某个$digest是否已经在运行了，因为它不想影响到被列入计划将要执行的那个。为此，Angular的作用域实现了一种叫做阶段（phase）的东西，它就是作用域上一个简单的字符串属性，存储了现在正在做的信息。

在Scope的构造函数里，我们引入一个叫$$phase的字段，初始化为null：

function Scope() {

this.$$watchers = [];

this.$$asyncQueue = [];

this.$$phase = null;

}

然后，我们定义一些方法用于控制这个阶段变量：一个用于设置，一个用于清除，也加个额外的检测，以确保不会把已经激活状态的阶段再设置一次：

Scope.prototype.$beginPhase = function(phase) {

if (this.$$phase) {

throw this.$$phase + ' already in progress.';

}

this.$$phase = phase;

};

Scope.prototype.$clearPhase = function() {

this.$$phase = null;

};

在$digest方法里，我们来从外层循环设置阶段属性为“$digest”：

Scope.prototype.$digest = function() {

var ttl = 10;

var dirty;

this.$beginPhase("$digest");

do {

while (this.$$asyncQueue.length) {

var asyncTask = this.$$asyncQueue.shift();

this.$eval(asyncTask.expression);

}

dirty = this.$$digestOnce();

if (dirty && !(ttl--)) {

this.$clearPhase();

throw "10 digest iterations reached";

}

} while (dirty);

this.$clearPhase();

};

我们把$apply也修改一下，在它里面也设置个跟自己一样的阶段。在调试的时候，这个会有些用：

Scope.prototype.$apply = function(expr) {

try {

this.$beginPhase("$apply");

return this.$eval(expr);

} finally {

this.$clearPhase();

this.$digest();

}

};

最终，把对$digest的调度放进$evalAsync。它会检测作用域上现有的阶段变量，如果没有（也没有已列入计划的异步任务），就把这个digest列入计划。

Scope.prototype.$evalAsync = function(expr) {

var self = this;

if (!self.$$phase && !self.$$asyncQueue.length) {

setTimeout(function() {

if (self.$$asyncQueue.length) {

self.$digest();

}

}, 0);

}

self.$$asyncQueue.push({scope: self, expression: expr});

};

有了这个实现之后，不管何时、何地，调用$evalAsync，都可以确定有一个digest会在不远的将来发生。

http://jsbin.com/iKeSaGi/1/embed?js,console

**在digest之后执行代码 - $$postDigest**

还有一种方式可以把代码附加到digest循环中，那就是把一个$$postDigest函数列入计划。

在Angular中，函数名字前面有双美元符号表示它是一个内部的东西，不是应用开发人员应该用的。但它确实存在，所以我们也要把它实现出来。

就像$evalAsync一样，$$postDigest也能把一个函数列入计划，让它“以后”运行。具体来说，这个函数将在下一次digest完 成之后运行。将一个$$postDigest函数列入计划不会导致一个digest也被延后，所以这个函数的执行会被推迟到直到某些其他原因引起一次 digest。顾名思义，$$postDigest函数是在digest之后运行的，如果你在$$digest里面修改了作用域，需要手动调 用$digest或者$apply，以确保这些变更生效。

首先，我们给Scope的构造函数加队列，这个队列给$$postDigest函数用：

function Scope() {

this.$$watchers = [];

this.$$asyncQueue = [];

this.$$postDigestQueue = [];

this.$$phase = null;

}

然后，我们把$$postDigest也加上去，它所做的就是把给定的函数加到队列里：

Scope.prototype.$$postDigest = function(fn) {

this.$$postDigestQueue.push(fn);

};

最终，在$digest里，当digest完成之后，就把队列里面的函数都执行掉。

Scope.prototype.$digest = function() {

var ttl = 10;

var dirty;

this.$beginPhase("$digest");

do {

while (this.$$asyncQueue.length) {

var asyncTask = this.$$asyncQueue.shift();

this.$eval(asyncTask.expression);

}

dirty = this.$$digestOnce();

if (dirty && !(ttl--)) {

this.$clearPhase();

throw "10 digest iterations reached";

}

} while (dirty);

this.$clearPhase();

while (this.$$postDigestQueue.length) {

this.$$postDigestQueue.shift()();

}

};

下面是关于如何使用$$postDigest函数的：

http://jsbin.com/IMEhowO/1/embed?js,console

**异常处理**

现有对Scope的实现已经逐渐接近在Angular中实际的样子了，但还有些脆弱，因为我们迄今为止没有花精力在异常处理上。

Angular的作用域在遇到错误的时候是非常健壮的：当产生异常的时候，不管在监控函数中，在$evalAsync函数中，还是在$$postDigest函数中，都不会把digest终止掉。我们现在的实现里，在以上任何地方产生异常都会把整个$digest弄挂。

我们可以很容易修复它，把上面三个调用包在try...catch中就好了。

Angular实际上是把这些异常抛给了它的$exceptionHandler服务。既然我们现在还没有这东西，先扔到控制台上吧。

$evalAsync和$$postDigest的异常处理是在$digest函数里，在这些场景里，从已列入计划的程序中抛出的异常将被记录成日志，它后面的还是正常运行：

Scope.prototype.$digest = function() {

var ttl = 10;

var dirty;

this.$beginPhase("$digest");

do {

while (this.$$asyncQueue.length) {

try {

var asyncTask = this.$$asyncQueue.shift();

this.$eval(asyncTask.expression);

} catch (e) {

(console.error || console.log)(e);

}

}

dirty = this.$$digestOnce();

if (dirty && !(ttl--)) {

this.$clearPhase();

throw "10 digest iterations reached";

}

} while (dirty);

this.$clearPhase();

while (this.$$postDigestQueue.length) {

try {

this.$$postDigestQueue.shift()();

} catch (e) {

(console.error || console.log)(e);

}

}

};

监听器的异常处理放在$$digestOnce里。

Scope.prototype.$$digestOnce = function() {

var self = this;

var dirty;

\_.forEach(this.$$watchers, function(watch) {

try {

var newValue = watch.watchFn(self);

var oldValue = watch.last;

if (!self.$$areEqual(newValue, oldValue, watch.valueEq)) {

watch.listenerFn(newValue, oldValue, self);

dirty = true;

}

watch.last = (watch.valueEq ? \_.cloneDeep(newValue) : newValue);

} catch (e) {

(console.error || console.log)(e);

}

});

return dirty;

};

现在我们的digest循环碰到异常的时候健壮多了。

http://jsbin.com/IMEhowO/2/embed?js,console

**销毁一个监听器**

当注册一个监听器的时候，一般都需要让它一直存在于整个作用域的生命周期，所以很少会要显式把它移除。也有些场景下，需要保持作用域的存在，但要把某个监听器去掉。

Angular中的$watch函数是有返回值的，它是个函数，如果执行，就把刚注册的这个监听器销毁。想在我们这个版本里实现这功能，只要返回一个函数在里面把这个监控器从$$watchers数组去除就可以了：

Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {

var self = this;

var watcher = {

watchFn: watchFn,

listenerFn: listenerFn,

valueEq: !!valueEq

};

self.$$watchers.push(watcher);

return function() {

var index = self.$$watchers.indexOf(watcher);

if (index >= 0) {

self.$$watchers.splice(index, 1);

}

};

};

现在我们就可以把$watch的这个返回值存起来，以后调用它来移除这个监听器：

http://jsbin.com/IMEhowO/4/embed?js,console

**展望未来**

我们已经走了很长一段路了，已经有了一个完美可以运行的类似Angular这样的脏检测作用域系统的实现了，但是Angular的作用域上面还做了更多东西。

或许最重要的是，在Angular里，作用域并不是孤立的对象，作用域可以继承于其他作用域，监听器也不仅仅是监听本作用域上的东西，还可以监听这 个作用域的父级作用域。这种方法，概念上很简单，但是对于初学者经常容易造成混淆。所以，本系列的下一篇文章主题就是作用域的继承。

后面我们会讨论Angular的事件系统，也是实现在Scope上的。