关于==和===的区别：http://stackoverflow.com/questions/359494/does-it-matter-which-equals-operator-vs-i-use-in-javascript-comparisons

The identity (`===`) operator behaves identically to the equality (`==`) operator except no type conversion is done, and the types must be the same to be considered equal.

Reference: Javascript Tutorial: Comparison Operators

The `==` operator will compare for equality *after doing any necessary type conversions*. The `===` operator will **not** do the conversion, so if two values are not the same type `===` will simply return `false`. It's this case where `===` will be faster, and may return a different result than `==`. In all other cases performance will be the same.

容易错的地方：

```
var a = [1,2,3];
var b = [1,2,3];

var c = { x: 1, y: 2 };
var d = { x: 1, y: 2 };

var e = "text";
var f = "te" + "xt";

a == b              // false
a === b             // false

c == d              // false
c === d             // false

e == f              // true
e === f             // true
```

new String(“abc”)得到的是一个对象，String {0: "a", 1: "b", 2: "c", length: 3, [[PrimitiveValue]]: "abc"}

第七章：Operator Expressions

Unary Operators，包括+，-，!。

先从+入手，和Javascript的区别是对于undefined变量，Angular是当作0处理而不是NaN。1，增加AST的node type，’UnaryExpression’。

```
AST.prototype.unary = function(){
    if (this.expect('+')){
        return {
            type: AST.UnaryExpression,
            operator: '+',
            argument: this.primary()
        };
    }else{
        return this.primary();
    }
};
```

2，Lexer中增加对符号的读取处理，而不是识别为unexpected next character。

然后处理!，同理。但是!符号要考虑它可能会是一个text，而不是operator，所以需要区分处理，即Lexer的readString方法中，push的是

```
this.tokens.push({
    text: rawString,
    value: string
});
```

Unary operators就这些了。

Multiplicative Operators，包括了 *,/,%。对符号的读取处理和Unary operator一样，assignment的左右改成this.multiplicative()，然后在compiler里面增加binary的处理。

在符号的解析中优先级比较重要，现在顺序是Primary<Unary<Multiplicative<Assignment。

现在如果有多个Multiplicative如 36*2%5，是会忽略掉一个operator之后的内容的。所以multiplicative方法中的if改成 while 循环

然后处理加减法操作，AST增加新的方法additive。

然后处理Relational and equality operators，处理>,<,<=,>=，以及==, !=，===，判断出true或者false。

接着处理relational and equality，正确的优先级顺序：

1. 2=="2">2==="2"   2. 2 == false === "2"   3. false === "2"   4. false

1. 2 + 3 < 6 - 2   2. 5 < 4   3. false

Lexer读符号的时候不再是只考虑一位，需要考虑==, ===的情况，因此else处理的时每次读符号需要考虑1，2，3位三种情况

然后处理AND, OR operator，An interesting detail about logical operators is that they are short-circuited. AND && 符号左边如果是false，右边则不会被执行。对应的测试例子：

```
it( short-circuits AND , function() {
var invoked;
var scope = {fn: function() { invoked = true; }};

  parse( false && fn() )(scope);
  expect(invoked).toBeUndefined();
});
```

同样，OR || 符号左边如果是True，右边也不会执行。AND比OR的优先级高。

```
case AST.LogicalExpression:
    intoId = this.nextId();
    this.state.body.push(this.assign(intoId,
this.recurse(ast.left)));
    this.if_(ast.operator === '&&'? intoId:
this.not(intoId), this.assign(intoId,
this.recurse(ast.right)));
    return intoId;
```

最后处理Ternary Operator，..?..:..。Lexer先移除?符号，然后分别解释为赋值语句，test?consequent:alternate，用两个this.if_完成处理。

优先级顺序：

1. Primary expressions: Lookups, function calls, method calls.
2. Unary expressions: +a, -a, !a.
3. Multiplicative arithmetic expressions: a * b, a / b, and a % b.
4. Additive arithmetic expressions: a + b and a - b.
5. Relationalexpressions:a<b,a>b,a<=b,anda>=b.
6. Equality testing expressions: a == b, a != b, a === b, and a !== b. 7. Logical AND expressions: a && b.
8. Logical OR expressions: a || b.
9. Ternary expressions: a ? b : c.
10. Assignments: a = b.

对优先级还需要做一些调整，AST.prototype.primary的if else语句先expect (，这样会最先处理括号。然后还需要处理多个语句的情况，这个时候返回值是最后一个语句，如：

```
it( returns the value of the last statement , function() { expect(parse( a = 1; b = 2; a + b )
({})).toBe(3);
});
```

总结：

1. AngularJS的各个expression支持的operator的实现
2. 优先级在AST中的实现
3. Angular的arithmetic expressions比Javascript更加forgiving
4. 对多个statements的支持

第八章：Filters，只有AngularJS有的feature，Javascript不包含

filter和plain function的区别：filters不需要被绑定在scopes上才能调用。

filter service：实现register 和 filter函数，

register：绑定name和factory function，实现方法是绑定在一个storage object上。如果是多个name, factory，则判断_.isObject(name)，如果是则_.map(name, function(factory, name)….

暂时filter service先实现这些，之后有了dependency injection system再讨论

Filter Expressions：解析如 parse(‘aString | upcase’)，先识别’|’符号，然后新建filter函数，

```
AST.prototype.filter = function() { var left = this.assignment();
if (this.expect( | )) {
left = {
type: AST.CallExpression, callee: this.identifier(), arguments: [left], filter: true
}; }
return left; };
```

然后在compiler中，如果有AST.CallExpression节点的filter=true，则返回callee(args)的形式。

```
ASTCompiler.prototype.filter = function(name){
    var filterId = this.nextId(true);
    if (!this.state.filters.hasOwnProperty('name')){
        this.state.filters[name] = filterId;
        return filterId;
    }
    return this.state.filters[name];
};

ASTCompiler.prototype.filterPrefix = function(){
    if (_.isEmpty(this.state.filters)){
        return '';
    }else{
        var prefix_parts = _.map(this.state.filters,
function(varName, filterName){
            return
varName+'=filter('+this.escape(filterName)+')';
        }, this);
        return 'var '+prefix_parts.join(',')+';';
    }
};
```

filter函数将callee的信息存在this.state.filters内，当AST被recurse之后，state.filters会包含所有expression中用到的filter。filter函数也需要在runtime被用到，和之前的ensureSafe...一样处理。

Filter chain expressions：AST.prototype.filter中的if (this.expect(‘|’)) 改成 while 就可以了

filters additional arguments：filter可以有多个参数，如：

```
it('can pass an additional argument to filters',
function() {
    register('repeat', function() {
        return function(s, times) {
            return _.repeat(s, times);
        };
    });
    var fn = parse('"hello" | repeat:3');
    expect(fn()).toEqual('hellohellohello');
});
```

在AST.prototype中增加对多个args的处理，while (this.expect(‘:’))…，args中push this.assignment()

The Filter Filter：

新建filter_filter.js，在filter.js中注册filter filter，filterFilter返回一个函数，参数是array, filterExpr

如果filterExpr是函数则直接使用lodash的filter函数，如果是字符串则创建一个predicate function

filter对象还可以是object，那么就需要deepCompare，比较object内部的value。

```
function deepCompare(actual, expected, comparator){ //
deep compare, concerning object
    if (_.isObject(actual)) {
        return _.some(actual, function(value){
//Checks if predicate returns truthy for any element
of collection
            return deepCompare(value, expected,
comparator);
        });
    }else{
        return comparator(actual, expected);
    }
}
```

对object用了_.some，只要有正例就返回True

Filer expression也可能是数字或者布尔值，用_.isNumber或者_.isBoolean判断，分别建立对应的createPredicateFn。对Null和Undefined在comparator中分别处理，如果actual 和 expect都是Null则返回actual === expected

Negated Filtering：filter string 前面加一个!，作为反选。在deepCompare中判断_.startsWith(expected, 1)，如果是!开头则返回!deepCompare(…)。

Filtering with object criteria: filter条件可以是一个对象，如

```
var fn = parse( arr | filter:{name: "o"} );
```

并且对象可以是nested object，也可以反选，如

```
var fn = parse( arr | filter:{name: {first: "!o"}} );
```

方法是loop over the expected criteria object, and for each value deep-compare the corresponding value in the actual object. 使用了_.every寻找满足所有条件的match，并且使用_.toPlainObject(object)处理对象，防止对prototype inheritance的所有inherited properties都进行检查。

Filtering With Object Wildcards：$代指任意属性，如

```
var fn = parse( arr | filter:{$: "o"} );
```

$可以表示出在nested object的哪一级，本身的filter object也可以是nested。实现方法是递归调用deepCompare的时候，给方法增加两个flag参数，表示是否matchAnyProperty，是否inWildCard。

Custom comparators：提供自定义的comparator function。createPredicateFn的时候判断comparator是否已有，并且_.isFunction，不是的话再定义comparator函数

comparator要求严格相等条件：增加一个布尔值表示是否严格相等，

```
var fn = parse( arr | filter:{name: "Jo"}:true );
```

如果comparator为true，直接使用_.isEqual即可。

总结：

1. filters的实质上是一个function call，用pipe operator |。Angular不支持bitwise operators
2. filters的注册和获得是通过filter service，绑定在一个stored object上
3. filter expressions被AST builder和compiler处理为call expressions，优先级最低
4. AST compiler来产生在runtime查找一个expression中所有用到的filters的代码
5. filter invocations可以是chained，也可以传递其他的参数
6. 内置的一些filter是如何工作的
7. wildcard $ keys，custom comparators