原型是JavaScript中一个比较难理解的概念，原型相关的属性也比较多，对象有"[[prototype]]"属性，函数对象有"prototype"属性，原型对象有"constructor"属性。

为了弄清原型，以及原型相关的这些属性关系，就有了这篇文章。

相信通过这篇文章一定能够清楚的认识到原型，现在就开始原型之旅吧。

认识原型

开始原型的介绍之前，首先来认识一下什么是原型？

在JavaScript中，原型也是一个对象，通过原型可以实现对象的属性继承，JavaScript的对象中都包含了一个" [[Prototype]]"内部属性，这个属性所对应的就是该对象的原型。

"[[Prototype]]"作为对象的内部属性，是不能被直接访问的。所以为了方便查看一个对象的原型，Firefox和Chrome中提供了"__proto__"这个非标准（不是所有浏览器都支持）的访问器（ECMA引入了标准对象原型访问器"Object.getPrototype(object)"）。

实例分析

下面通过一个例子来看看原型相关概念：

```
function Person(name, age){
    this.name = name;
    this.age = age;

    this.getInfo = function(){
        console.log(this.name + " is " + this.age + " years old");
    };
}

var will = new Person("Will", 28);
```

在上面的代码中，通过了Person这个构造函数创建了一个will对象。下面就通过will这个对象一步步展开了解原型。

Step 1： 查看对象**will**的原型

通过下面代码，可以查看对象will的原型：

```
console.log(will.__proto__);
console.log(will.constructor);
```

结果分析:

- "Person {}"对象就是对象will的原型，通过Chrome展开可以看到，"Person {}"作为一个原型对象，也有"__proto__"属性（对应原型的原型）。
- 在这段代码中，还用到了"constructor"属性。在**JavaScript**的原型对象中，还包含一个**"constructor"**属性，这个属性对应创建所有指向该原型的实例的构造函数。
    - 通过"constructor"这个属性，我们可以来判断一个对象是不是数组类型

      ```
      function isArray(myArray) {
          return myArray.constructor.toString().indexOf("Array") > -1;
      }
      ```
    - 在这里，**will**对象本身并没有**"constructor"**这个属性，但是通过原型链查找，找到了will原型（will.__proto__）的"constructor"属性，并得到了Person函数。

```
> will.__proto__
◂ ▼Person {} i
    ▼constructor: function Person(name, age)
       arguments: null
       caller: null
       length: 2
       name: "Person"
      ▶prototype: Person
      ▶__proto__: function ()
      ▶<function scope>
    ▶__proto__: Object
> will.constructor
◂ function Person(name, age){
      this.name = name;
      this.age = age;

      this.getInfo = function(){
          console.log(this.name + " is " + this.age + " years old");
      };
  }
```

Step 2： 查看对象**will**的原型（**will.__proto__**）的原型

既然will的原型"Person {}"也是一个对象，那么我们就同样可以来查看"will的原型（will.__proto__）的原型"。

运行下面的代码：

```
console.log(will.__proto__ === Person.prototype);
console.log(Person.prototype.__proto__);
console.log(Person.prototype.constructor);
console.log(Person.prototype.constructor === Person);
```

结果分析：

- 首先看 "will.__proto__ === Person.prototype"，在**JavaScript**中，每个函数都有一个**prototype**属性，当一个函数被用作构造函数来创建实例时，该函数的**prototype**属性值将被作为原型赋值给所有对象实例（也就是设置实例的**__proto__**属性），也就是说，所有实例的原型引用的是函数的**prototype**属性。了解了构造函数的prototype属性之后，一定就明白为什么第一句结果为true了。
    - prototype属性是函数对象特有的，如果不是函数对象，将不会有这样一个属性。
- 当通过"Person.prototype.__proto__"语句获取will对象原型的原型时候，将得到"Object {}"对象，后面将会看到所有对象的原型都将追溯到"Object {}"对象。
- 对于原型对象"Person.prototype"的"constructor"，根据前面的介绍，将对应Person函数本身。

通过上面可以看到，**"Person.prototype"**对象和**Person**函数对象通过**"constructor"**和**"prototype"**属性实现了相互引用（后面会有图展示这个相互引用的关系）。

```
> will.__proto__ === Person.prototype
< true
> Person.prototype.__proto__
< ▶ Object {}
> Person.prototype.constructor
< function Person(name, age){
      this.name = name;
      this.age = age;

      this.getInfo = function(){
          console.log(this.name + " is " + this.age + " years old");
      };
  }
> Person.prototype.constructor === Person
< true
```

Step 3: 查看对象**Object**的原型

通过前一部分可以看到，will的原型的原型是"Object {}"对象。实际上在JavaScript中，所有对象的原型都将追溯到"Object {}"对象。

下面通过一段代码看看"Object {}"对象：

```
console.log(Person.prototype.__proto__ === Object.prototype);
console.log(typeof Object);
console.log(Object);
console.log(Object.prototype);
console.log(Object.prototype.__proto__);
console.log(Object.prototype.constructor);
```

通过下面的代码可以看到：

- Object对象本身是一个函数对象。
- 既然是Object函数，就肯定会有prototype属性，所以可以看到"Object.prototype"的值就是"Object {}"这个原型对象。
- 反过来，当访问"Object.prototype"对象的"constructor"这个属性的时候，就得到了Obejct函数。
- 另外，当通过"Object.prototype.__proto__"获取Object原型的原型的时候，将会得到"null"，也就是说"Object {}"原型对象就是原型链的终点了。

```
> Person.prototype.__proto__ === Object.prototype
< true
> typeof Object
< "function"
> Object
< function Object() { [native code] }
> Object.prototype
< ▶ Object {}
> Object.prototype.__proto__
< null
> Object.prototype.constructor
< function Object() { [native code] }
```

Step 4: 查看对象**Function**的原型

在上面的例子中，Person是一个构造函数，在JavaScript中函数也是对象，所以，我们也可以通过"__proto__"属性来查找Person函数对象的原型。

```
console.log(Person.__proto__ === Function.prototype);
console.log(Person.constructor === Function)
console.log(typeof Function);
console.log(Function);
console.log(Function.prototype);
console.log(Function.prototype.__proto__);
console.log(Function.prototype.constructor);
```

结果分析：

- 在JavaScript中有个Function对象（类似Object），这个对象本身是个函数；所有的函数（包括Function，Object）的原型（__proto__）都是"Function.prototype"。
- Function对象作为一个函数，就会有prototype属性，该属性将对应"function () {}"对象。
- Function对象作为一个对象，就有"__proto__"属性，该属性对应"Function.prototype"，也就是说，"Function.__proto__ === Function.prototype"
- 对于Function的原型对象"Function.prototype"，该原型对象的"__proto__"属性将对应"Object {}"

```
> Person.__proto__ === Function.prototype
< true
> Person.constructor === Function
< true
> typeof Function
< "function"
> Function
< function Function() { [native code] }
> Function.prototype
< function () {}
> Function.prototype.__proto__
< ▶ Object {}
> Function.prototype.constructor
< function Function() { [native code] }
```

对比**prototype**和**__proto__**

对于"prototype"和"__proto__"这两个属性有的时候可能会弄混，"Person.prototype"和"Person.__proto__"是完全不同的。

在这里对"prototype"和"__proto__"进行简单的介绍：

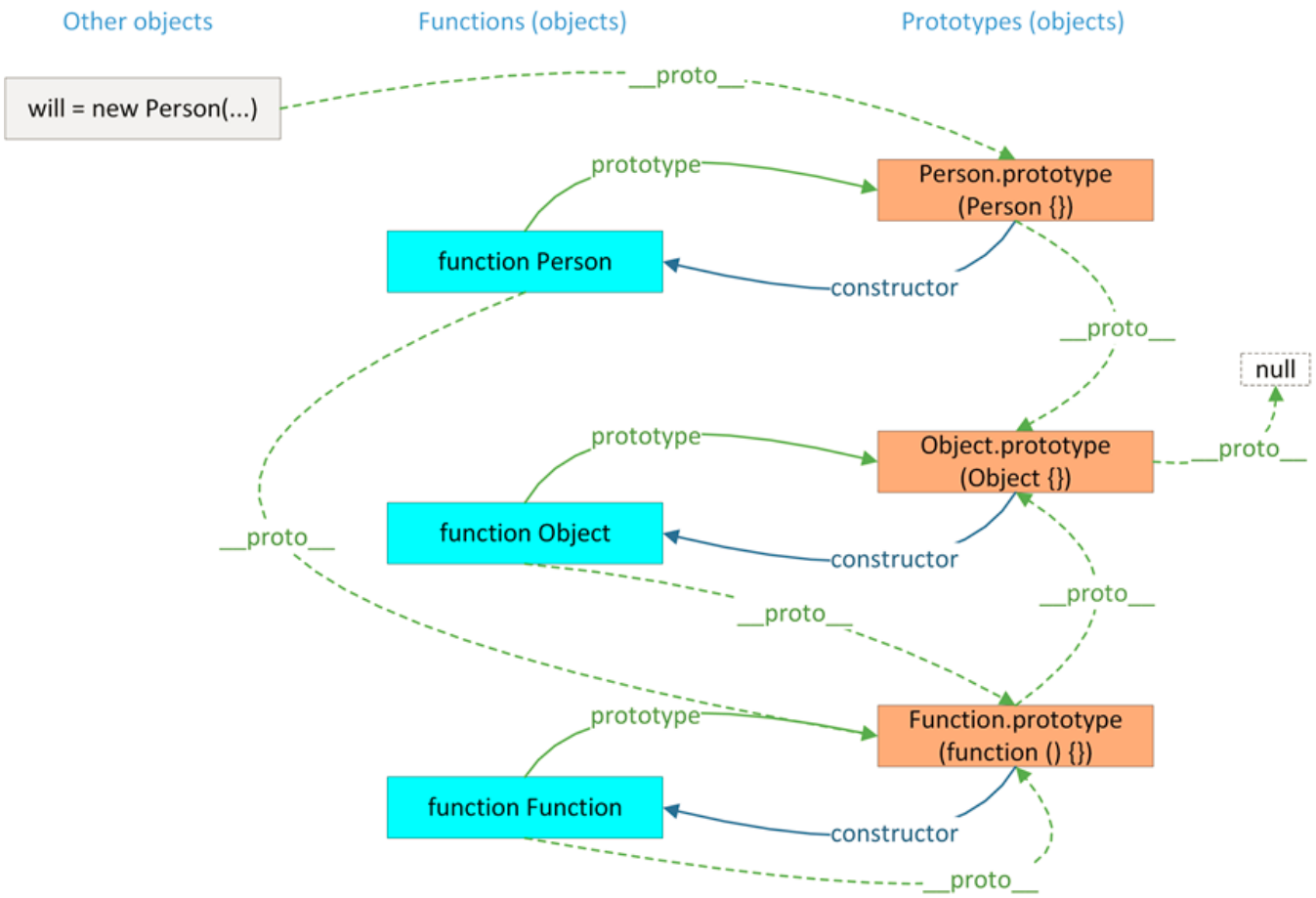
- 对于所有的对象，都有__proto__属性，这个属性对应该对象的原型
- 对于函数对象，除了__proto__属性之外，还有prototype属性，当一个函数被用作构造函数来创建实例时，该函数的**prototype**属性值将被作为原型赋值给所有对象实例（也就是设置实例的**__proto__**属性）

图解实例

通过上面结合实例的分析，相信你一定了解了原型中的很多内容。

但是现在肯定对上面例子中的关系感觉很凌乱，一会儿原型，一会儿原型的原型，还有Function，Object，constructor，prototype等等关系。

现在就对上面的例子中分析得到的结果/关系进行图解，相信这张图可以让你豁然开朗。



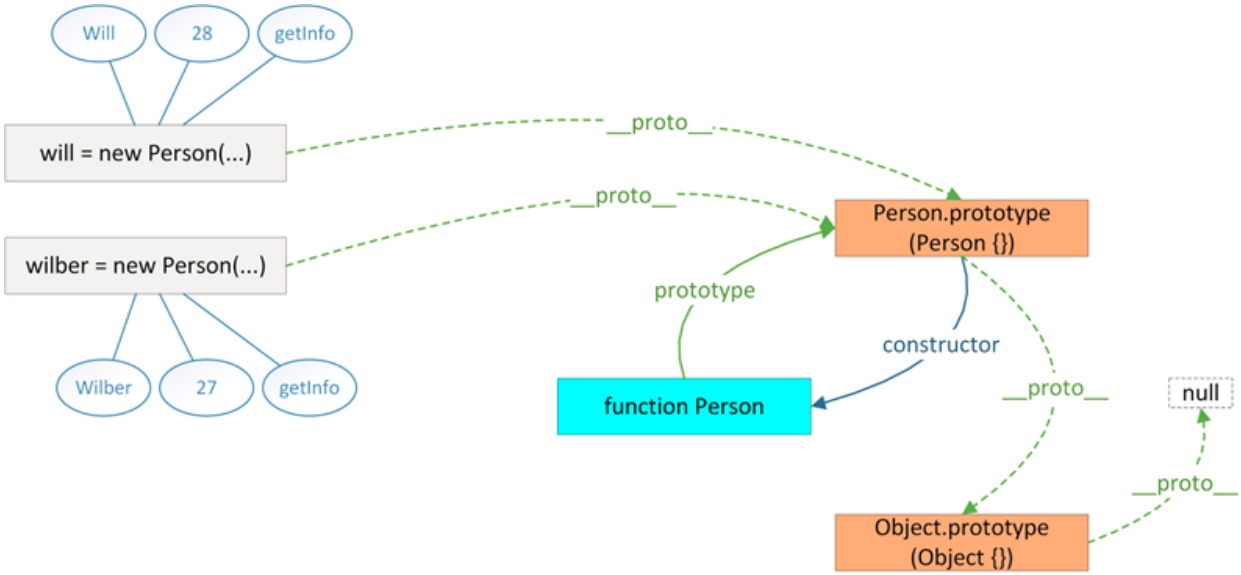
对于上图的总结如下：

- 所有的对象都有"__proto__"属性，该属性对应该对象的原型
- 所有的函数对象都有"prototype"属性，该属性的值会被赋值给该函数创建的对象的"__proto__"属性
- 所有的原型对象都有"constructor"属性，该属性对应创建所有指向该原型的实例的构造函数
- 函数对象和原型对象通过"prototype"和"constructor"属性进行相互关联

通过原型改进例子

在上面例子中，"getInfo"方法是构造函数Person的一个成员，当通过Person构造两个实例的时候，每个实例都会包含一个"getInfo"方法。

```
var will = new Person("Will", 28);
var wilber = new Person("Wilber", 27);
```



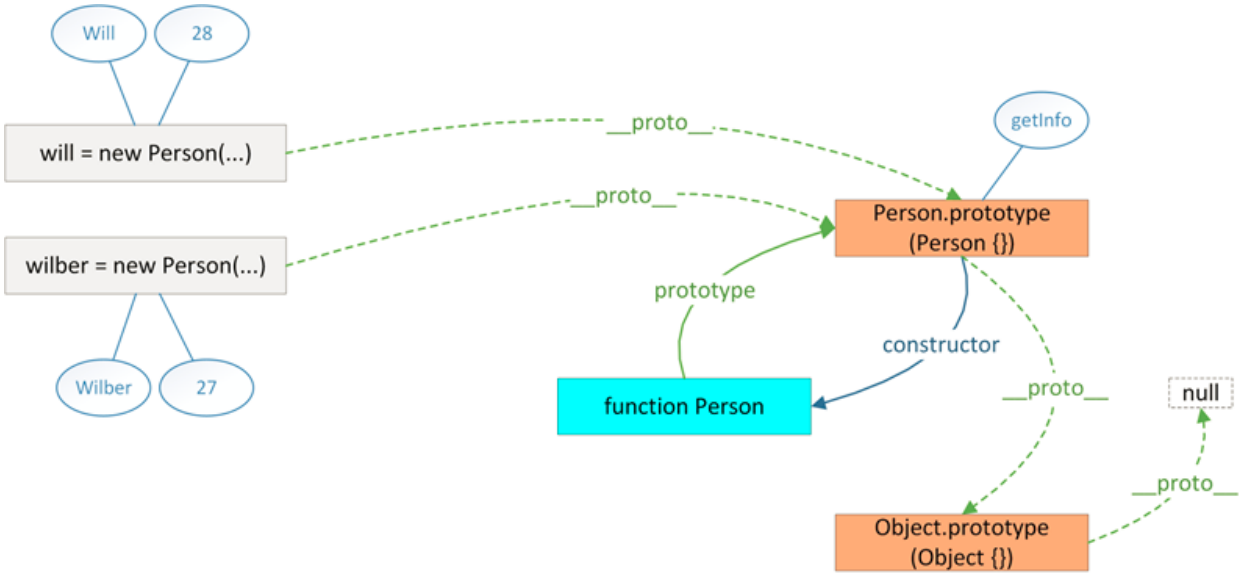
前面了解到，原型就是为了方便实现属性的继承，所以可以将"getInfo"方法当作Person原型（Person.__proto__）的一个属性，这样所有的实例都可以通过原型继承的方式来使用"getInfo"这个方法了。

所以对例子进行如下修改：

```
function Person(name, age){
    this.name = name;
    this.age = age;
}

Person.prototype.getInfo = function(){
    console.log(this.name + " is " + this.age + " years old");
};
```

修改后的结果为：



原型链

因为每个对象和原型都有原型，对象的原型指向对象的父，而父的原型又指向父的父，这种原型层层连接起来的就构成了原型链。

在"理解JavaScript的作用域链"一文中，已经介绍了标识符和属性通过作用域链和原型链的查找。

这里就继续看一下基于原型链的属性查找。

属性查找

当查找一个对象的属性时，JavaScript 会向上遍历原型链，直到找到给定名称的属性为止，到查找到达原型链的顶部（也就是"Object.prototype"），如果仍然没有找到指定的属性，就会返回 undefined。

看一个例子：

```
function Person(name, age){
    this.name = name;
    this.age = age;
}

Person.prototype.MaxNumber = 9999;
Person.__proto__.MinNumber = -9999;

var will = new Person("Will", 28);

console.log(will.MaxNumber);
// 9999
console.log(will.MinNumber);
// undefined
```

在这个例子中分别给"Person.prototype "和" Person.__proto__"这两个原型对象添加了"MaxNumber "和"MinNumber"属性，这里就需要弄清"prototype"和"__proto__"的区别了。

"Person.prototype "对应的就是Person构造出来所有实例的原型，也就是说"Person.prototype "属于这些实例原型链的一部分，所以当这些实例进行属性查找时候，就会引用到"Person.prototype "中的属性。

属性隐藏

当通过原型链查找一个属性的时候，首先查找的是对象本身的属性，如果找不到才会继续按照原型链进行查找。

这样一来，如果想要覆盖原型链上的一些属性，我们就可以直接在对象中引入这些属性，达到属性隐藏的效果。

看一个简单的例子：

```
function Person(name, age){
    this.name = name;
    this.age = age;
}

Person.prototype.getInfo = function(){
    console.log(this.name + " is " + this.age + " years old");
};

var will = new Person("Will", 28);
will.getInfo = function(){
    console.log("getInfo method from will instead of prototype");
};

will.getInfo();
// getInfo method from will instead of prototype
```

对象创建方式影响原型链

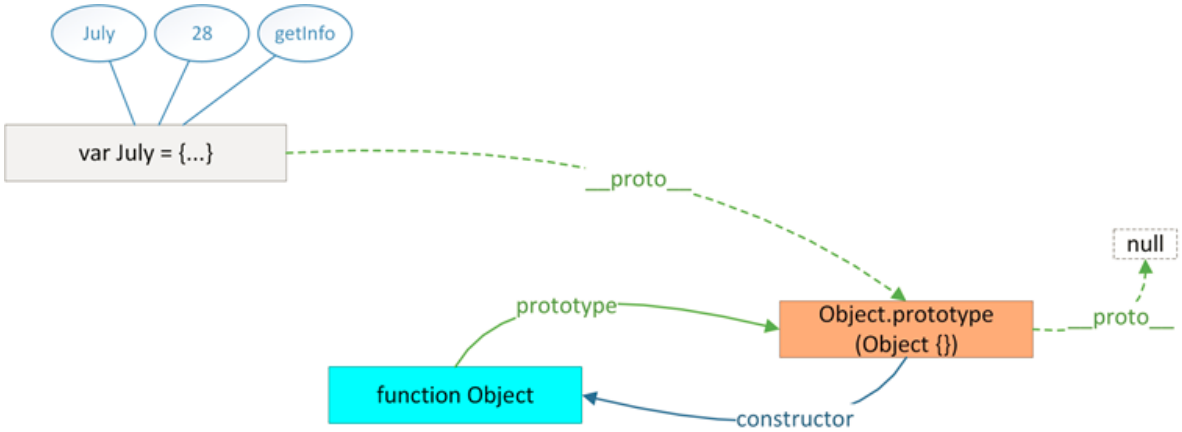
会到本文开始的例子，will对象通过Person构造函数创建，所以will的原型（will.__proto__）就是"Person.prototype"。

同样，我们可以通过下面的方式创建一个对象：

```
var July = {
    name: "July",
    age: 28,
    getInfo: function(){
        console.log(this.name + " is " + this.age + " years old");
    },
}

console.log(July.getInfo());
```

当使用这种方式创建一个对象的时候，原型链就变成下图了，July对象的原型是"Object.prototype"也就是说对象的构建方式会影响原型链的形式。



hasOwnProperty

"hasOwnProperty"是"Object.prototype"的一个方法，该方法能判断一个对象是否包含自定义属性而不是原型链上的属性，因为"hasOwnProperty" 是 JavaScript 中唯一一个处理属性但是不查找原型链的函数。

相信你还记得文章最开始的例子中，通过will我们可以访问"constructor"这个属性，并得到will的构造函数Person。这里结合"hasOwnProperty"这个函数就可以看到，will对象并没有"constructor"这个属性。

从下面的输出可以看到，"constructor"是will的原型（will.__proto__）的属性，但是通过原型链的查找，will对象可以发现并使用"constructor"属性。

```
> will.constructor
< function Person(name, age){
      this.name = name;
      this.age = age;
  }
> will.hasOwnProperty("constructor")
< false
> will.__proto__.hasOwnProperty("constructor")
< true
```

"hasOwnProperty"还有一个重要的使用场景，就是用来遍历对象的属性。

```
function Person(name, age){
    this.name = name;
    this.age = age;
}

Person.prototype.getInfo = function(){
    console.log(this.name + " is " + this.age + " years old");
};


var will = new Person("Will", 28);

for(var attr in will){
    console.log(attr);
}
// name
// age
// getInfo

for(var attr in will){
    if(will.hasOwnProperty(attr)){
        console.log(attr);
    }
}
// name
// age
```

总结

本文介绍了JavaScript中原型相关的概念，对于原型可以归纳出下面一些点：

- 所有的对象都有"[[prototype]]"属性（通过__proto__访问），该属性对应对象的原型
- 所有的函数对象都有"prototype"属性，该属性的值会被赋值给该函数创建的对象的"__proto__"属性
- 所有的原型对象都有"constructor"属性，该属性对应创建所有指向该原型的实例的构造函数
- 函数对象和原型对象通过"prototype"和"constructor"属性进行相互关联

还有要强调的是文章开始的例子，以及通过例子得到的一张"普通对象"，"函数对象"和"原型对象"之间的关系图，当你对原型的关系迷惑的时候，就想想这张图（或者重画一张当前对象的关系图），就可以理清这里面的复杂关系了。

通过这些介绍，相信一定可以对原型有个清晰的认识。