Auto Layout：Snapkit源码剖析

Snapkit是目前Swift中通过代码进行Auto Layout布局时最流行的开源库。与OC中最主流的Auto Layout开源库Masonry是同一个团队维护，有着相似的API风格。

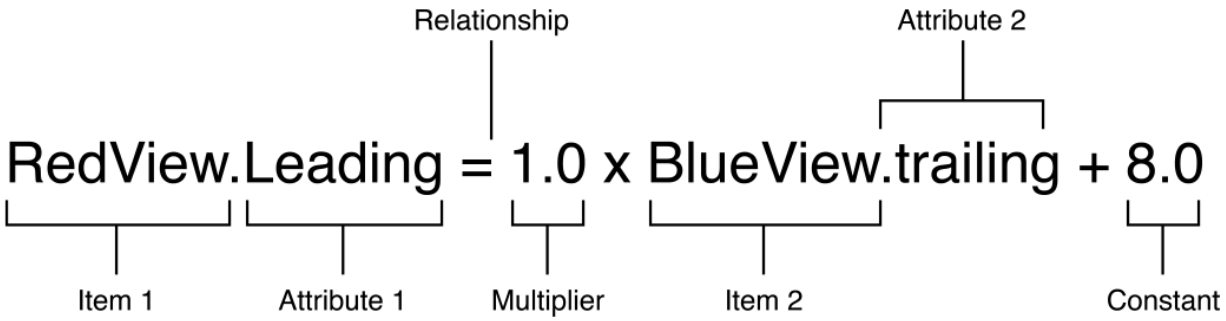
Auto Layout的本质

Auto Layout可以理解为描述一个界面中各个元素之间布局的关系的一种语言。每个关系描述对应的是一条约束。

平面关系里两个点的关系可以用一次函数（y = ax + b）来表示，与此相似，每条约束的定义方式与一次函数也有着一样的参数：

> item1.attribute1 = multiplier × item2.attribute2 + constant

再贴一张官方的示意图：



这条约束表示RedView的左边与BlueView右边的距离为8。

原生API的问题

Auto Layout从编码层面看本质就是约束（NSLayoutConstraint），通过给一个元素声明约束来表示它的布局关系。

Visual Format Language

Visual Format Language是创建约束的一种方式。通过一句符合指定语法的字符串来生成约束。

写起来的风格就像这样：

```
let viewsDictionary = ["label1": label1, "label2": label2, "label3": label3,
"label4": label4]
let metrics = ["labelHeight": 88]
let constraint = NSLayoutConstraint.constraints(withVisualFormat:
"V:|[label1(labelHeight)]-[label2(labelHeight)]-[label3(labelHeight)]-
[label4(labelHeight)]-(>=10)-|",
options: [], metrics: metrics, views: viewsDictionary)
view.addConstraints(constraint)
```

有着自己的一套用于表示相关元素关系的语法。

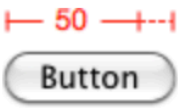
下面截图展示了一些VFL的语法：

Standard Space

[button]-[textField]

Width Constraint

[button(>=50)]



Connection to Superview

|-50-[purpleBox]-50-|

优点

- Xcode控制台输出约束时使用的就是这种语法。如果对这种语法很熟悉，调试约束时会有很大帮助。
- 一次可以表示几个物体在一个方向上的约束关系，所以可以一次创建几个约束。
- 在这种语法限制下只能创建出有效的约束。

缺点

- VFL的语法适合表述平面关系的约束，对于一些约束，比如比例（aspect ratios）相关的约束无法通过VFL创建。
- 编译器不会检查VFL的字符串，所以只能在运行时调试约束。

实际使用：不推荐

VFL通过一个约定格式的字符串表示约束关系。在实际项目中，一个无法被检查的长字符串很容易发生拼写错误。VFL的语法也有一定的学习门槛，与我们自然语言中描述约束关系的方式相差很远。约束通过一个字符串表示也导致了不好复用的问题。所以在实际项目中通常不会选择VFL。

iOS 7 & 8: NSLayoutConstraint

假设我们要对blueView布局为长宽都为100，在iOS 7中这样写：

```
blueView.translatesAutoresizingMaskIntoConstraints = false

// iOS 7
let widthConstraint = NSLayoutConstraint(item: blueView, attribute: .width,
relatedBy: .equal, toItem: nil, attribute: .notAnAttribute, multiplier: 1,
constant: 100)
blueView.addConstraint(widthConstraint)
let heightConstraint = NSLayoutConstraint(item: blueView, attribute:
.height, relatedBy: .equal, toItem: nil, attribute: .notAnAttribute,
multiplier: 1, constant: 100)
blueView.addConstraint(heightConstraint)
```

接着再设置水平竖直居中，在iOS 8中这样写：

```
// iOS 8
let centerXConstraint = NSLayoutConstraint(item: blueView, attribute:
.centerX, relatedBy: .equal, toItem: view, attribute: .centerX, multiplier:
1, constant: 0)
centerXConstraint.isActive = true

NSLayoutConstraint(item: blueView, attribute: .centerY, relatedBy: .equal,
toItem: view, attribute: .centerY, multiplier: 1, constant: 0).isActive =
true
```

创建`NSLayoutConstraint`对象的方法参数是一致的，区别只是在iOS 7中需要指出这条约束应该加在哪个对象上。在iOS 8中省去了这个判断，只需要设置这条约束的`isActive`为true时就可以。可以这样做是因为一条约束自身已经知道这条约束对应的是哪两个对象，再用代码表明这条约束应该加在哪个对象身上是不必须的。

评价

这样的API虽然完整的表达了约束所需要的参数，但是写法却非常繁琐。每条约束都需要7个参数，即使有的参数是不必须的。每次只能添加一条约束。通过上面的代码也可以直观的看出，只是设置一个view的简单约束代码已经一堆了，API的表现力很差。

iOS 9 : Layout Anchors

Auto Layout发布两年后，苹果推出了一套新的API来创建约束。我很怀疑苹果设计这组API参考了Snapkit，因为风格上有些接近。

和上面一节实现同样的布局，在iOS 9后代码这样写：

```
blueView.translatesAutoresizingMaskIntoConstraints = false
blueView.widthAnchor.constraint(equalToConstant: 100).isActive = true
blueView.heightAnchor.constraint(equalToConstant: 100).isActive = true
blueView.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive =
true
blueView.centerYAnchor.constraint(equalTo: view.centerYAnchor).isActive =
true
```

constraint函数利用了swift函数默认参数的语言特性，将multipier和constant配置了默认值。完整的函数签名如下：

```
func constraint(equalTo anchor: NSLayoutDimension, multiplier m: CGFloat,
constant c: CGFloat) -> NSLayoutConstraint
```

总体而言相比之前的写法有了一个巨大的进步。约束的添加逻辑更加接近我们思维的方式，友好很多。

Snapkit

没有对比就没有伤害。同样的界面如果用Snapkit写起来是这样的：

```
blueView.snp.makeConstraints { (make) in
            make.width.height.equalTo(100)
            make.center.equalToSuperview()
        }
```

总共只要3行代码，简洁清晰。

源码剖析

Attributes

先介绍一下最基础的attribute。

NSLayoutAttribute

系统API提供的属性是`NSLayoutAttribute`，是一个枚举：

```
enum NSLayoutAttribute : Int {
    case left
    case right
    case top
    //...
    case notAnAttribute
}
```

ConstraintAttributes

`NSLayoutAttribute`的不足之处在于一次只能表达一个属性。

然而有时我们几个属性都是对应一个表达式的值，比如下面的代码。

```
make.width.height.equalTo(100)
```

一个值可以对应多个选项，在swift中就是`OptionSet`协议了。

这里稍微展示一下`OptionSet`的用法：

```
struct ShippingOptions: OptionSet {
    let rawValue: Int

    static let nextDay    = ShippingOptions(rawValue: 1 << 0)
    static let secondDay  = ShippingOptions(rawValue: 1 << 1)
    static let priority   = ShippingOptions(rawValue: 1 << 2)
    static let standard   = ShippingOptions(rawValue: 1 << 3)

    static let express: ShippingOptions = [.nextDay, .secondDay]
    static let all: ShippingOptions = [.express, .priority, .standard]
}
```

上面的代码定义的`all`的就是一个选项的集合。我们在动画中经常用到的`UIViewAnimationOptions`也是这样的类型。

`ConstraintAttributes`是Snapkit中用于表示attribute的类型。定义了与`NSLayoutAttribute`一致的属性：

```
struct ConstraintAttributes : OptionSet {
    static var none: ConstraintAttributes { return self.init(0) }
    static var left: ConstraintAttributes { return self.init(1) }
    static var top: ConstraintAttributes {  return self.init(2) }
    static var right: ConstraintAttributes { return self.init(4) }
    //...
}
```

细节：重载 + 运算符

苹果在`OptionSet`中定义了几个集合运算的函数，为了更便捷的使用，Snapkit自定义了几个运算符：

```
func + (left: ConstraintAttributes, right: ConstraintAttributes) ->
ConstraintAttributes {
    return left.union(right)
}

func +=(left: inout ConstraintAttributes, right: ConstraintAttributes) {
    left.formUnion(right)
}

func -=(left: inout ConstraintAttributes, right: ConstraintAttributes) {
    left.subtract(right)
}

func ==(left: ConstraintAttributes, right: ConstraintAttributes) -> Bool {
    return left.rawValue == right.rawValue
}
```

如何转换为NSLayoutAttribute

因为最后调用的还是系统的API，所以`ConstraintAttributes`需要转换为对应的`NSLayoutAttribute`类型。这段逻辑放在计算属性`layoutAttributes`中：

```
var layoutAttributes: [NSLayoutAttribute] {
        var attrs = [NSLayoutAttribute]()
        if (self.contains(ConstraintAttributes.left)) {
            attrs.append(.left)
        }
        if (self.contains(ConstraintAttributes.top)) {
            attrs.append(.top)
        }
        if (self.contains(ConstraintAttributes.right)) {
            attrs.append(.right)
        }
        //...
        return attrs
}
```

每次获取`layoutAttributes`属性时，就会根据自身的值生成一个对应的`NSLayoutAttribute`数组返回。

简单的基础Model

Snapkit不仅支持iOS平台，还支持tvOS、macOS其他平台。为了更方便的支持多平台，Snapkit把几个常用对象都重新封装了一次。

ConstraintView

`ConstraintView`是一个别名，对应的就是iOS上的`UIView`：

```
#if os(iOS) || os(tvOS)
    public typealias ConstraintView = UIView
#else
    public typealias ConstraintView = NSView
#endif
```

其他几个别名

同上节，Snapkit还定义其他几个别名，对应关系如下表：

Snapkit	iOS
ConstraintLayoutGuide	UILayoutGuide
ConstraintLayoutSupport	UILayoutSupport
ConstraintInsets	UIEdgeInsets
ConstraintInterfaceLayoutDirection	UIUserInterfaceLayoutDirection

ConstraintRelation

猜测由于历史原因`ConstraintRelation`没有采用别名的方式，而是重新封装了一遍。

```
enum ConstraintRelation : Int {
    case equal = 1
    case lessThanOrEqual
    case greaterThanOrEqual

    var layoutRelation: NSLayoutRelation {
        get {
            switch(self) {
            case .equal:
                return .equal
            case .lessThanOrEqual:
                return .lessThanOrEqual
            case .greaterThanOrEqual:
                return .greaterThanOrEqual
            }
        }
    }
}
```

通过`layoutRelation`获取到对应的`NSLayoutRelation`值。

ConstraintItem

一条约束对应两个两个Item。

> item1.attribute1 = multiplier × item2.attribute2 + constant

一个Item包含两个属性：target和attributes，表示这个Item的实例和属性：

```
public final class ConstraintItem {
   weak var target: AnyObject?
   let attributes: ConstraintAttributes
   //...
}
```

同时注意到这个类被标记为了final，这样可以提高编译的性能，也表明了这个类不能被继承。

重载 == 运算符

```
public func ==(lhs: ConstraintItem, rhs: ConstraintItem) -> Bool {
    // pointer equality
    guard lhs !== rhs else {
        return true
    }

    // must both have valid targets and identical attributes
    guard let target1 = lhs.target,
          let target2 = rhs.target,
          target1 === target2 && lhs.attributes == rhs.attributes else {
            return false
    }

    return true
}
```

`ConstraintItem`还重载了 == 运算符。首先判断两个实例是不是指向同一块内存地址，注意这里是判断的运算符是`!==`。接着再判断两个Item的attributes是否一样。

layoutConstraintItem

target的类型是`AnyObject`，在进行逻辑判断时很不方便。专门暴露出了一个已经类型转换好的属性：

```
var layoutConstraintItem: LayoutConstraintItem? {
        return self.target as? LayoutConstraintItem
}
```

LayoutConstraintItem

`LayoutConstraintItem`是一个协议，用于表示可以添加约束的对象。在iOS 9之前，只有`UIView`能添加约束，在iOS 9中，引入了可以参与辅助布局的`UILayoutGuide`，可以和`UIView`一样添加约束。

```
public protocol LayoutConstraintItem: class {
}

@available(iOS 9.0, OSX 10.11, *)
extension ConstraintLayoutGuide : LayoutConstraintItem {
}

extension ConstraintView : LayoutConstraintItem {
}
```

那么这个协议定义那些函数和属性呢?

prepare

定义了一个`prepare`函数，如果是`UIView`对象，则把`translatesAutoresizingMaskIntoConstraints`设置为`false`：

```
extension LayoutConstraintItem {

    internal func prepare() {
        if let view = self as? ConstraintView {
            view.translatesAutoresizingMaskIntoConstraints = false
        }
    }

     //...
}
```

superview

比如居中的约束常常是对于`superview`而言，在协议里扩展了一个`superview`属性：

```
var superview: ConstraintView? {
        if let view = self as? ConstraintView {
            return view.superview
        }

        if #available(iOS 9.0, OSX 10.11, *), let guide = self as?
ConstraintLayoutGuide {
            return guide.owningView
        }

        return nil
    }
```

constraints

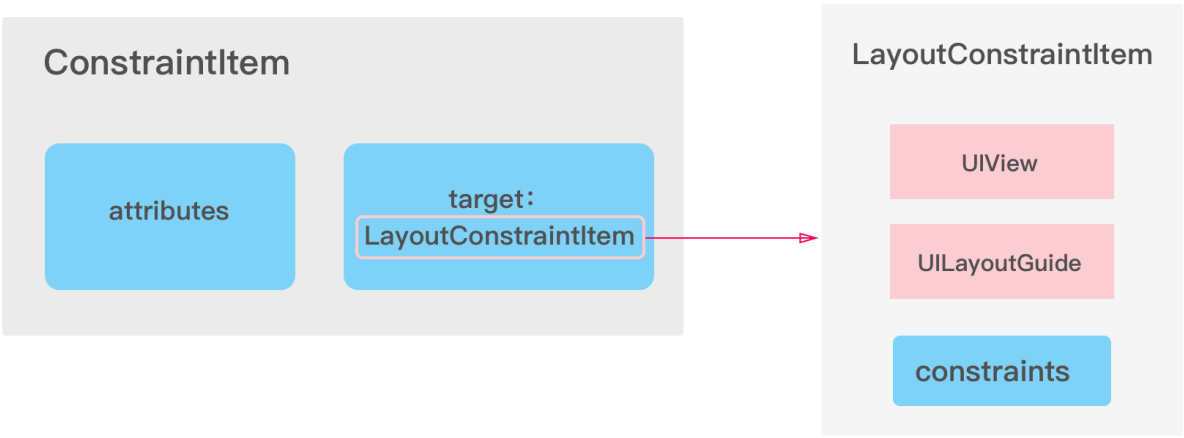
每个UIView身上都可能有几条约束，所以定义了几个添加移除约束的函数，和一个用于保存约束的集合：

```
 var constraints: [Constraint] {
      return self.constraintsSet.allObjects as! [Constraint]
 }

 func add(constraints: [Constraint]) {
       //...
 }

 func remove(constraints: [Constraint]) {
        //...
 }

 private var constraintsSet: NSMutableSet {
        let constraintsSet: NSMutableSet

        if let existing = objc_getAssociatedObject(self, &constraintsKey)
as? NSMutableSet {
            constraintsSet = existing
        } else {
            constraintsSet = NSMutableSet()
            objc_setAssociatedObject(self, &constraintsKey,
            constraintsSet,.OBJC_ASSOCIATION_RETAIN_NONATOMIC)
        }
        return constraintsSet
    }
```

为了保证相同的约束不会被重复添加，内部用了`NSMutableSet`来保存。为什么不是swift中的`Set`类型呢？因为这个集合定义在扩展里，通过OC中的runtime方法`objc_getAssociatedObject`保存，所以需要是OC的对象，于是选择了`NSMutableSet`。最后封装成了一个数组`constraints`暴露给外界。

图示

用一张图简单的表示就是这样：



ConstraintItem包含一组属性和一个对应的可布局实例（**LayoutConstraintItem**），这个实例可能是`UIView`也可能是`UILayoutGuide`，这个可布局实例身上的`constraints`表示自身已经持有的一组约束。

LayoutConstraint

`LayoutConstraint`继承自`NSLayoutConstraint`。为了保持自身命名风格的统一，将`identifier`属性封装成了`label`，换了一个属性名。

```
public class LayoutConstraint : NSLayoutConstraint {

    public var label: String? {
        get {
            return self.identifier
        }
        set {
            self.identifier = newValue
        }
    }

    weak var constraint: Constraint? = nil

}
```

Snapkit中一条约束可能对应几个`NSLayoutConstraint`，所以这里有一个weak的`constraint`属性指向生成这条约束的Snapkit约束实例。

*Target

看这样的代码：

```
make.width.height.equalTo(100)
make.center.equalTo(self.view)
```

注意到`equalTo`的参数可以是一个值，也可以是一个`UIView`类型。

在iOS 9的API则是笨拙一点的两个函数：

```
open func constraint(equalToConstant c: CGFloat) -> NSLayoutConstraint

open func constraint(equalTo anchor: NSLayoutAnchor<AnchorType>) ->
NSLayoutConstraint
```

Snapkit通过定义了一个*Target的协议来达到这个目的。这里列举一个简单的例子`ConstraintPriorityTarget`，这个协议就是指可以作为设置Priority的类型。

```
public protocol ConstraintPriorityTarget {

    var constraintPriorityTargetValue: Float { get }

}

extension Int: ConstraintPriorityTarget {

    public var constraintPriorityTargetValue: Float {
        return Float(self)
    }

}

// ....
extension Double: ConstraintPriorityTarget {

    public var constraintPriorityTargetValue: Float {
        return Float(self)
    }

}

extension CGFloat: ConstraintPriorityTarget {

    public var constraintPriorityTargetValue: Float {
        return Float(self)
    }

}
```

可以看到并没有什么太好的办法，在协议中定义了需要的类型*TargetValue，然后在每个实现了这个协议的类型中转换类型。这种方式虽然增加了一点代码的复杂度，但是对于使用API的用户而则省去了类型转换的步骤。

这样的*Target有以下几个：

协议	获取Value的属性、函数
ConstraintRelatableTarget	没有
ConstraintConstantTarget	constraintConstantTargetValueFor(layoutAttribute: NSLayoutAttribute)
ConstraintPriorityTarget	constraintPriorityTargetValue
ConstraintMultiplierTarget	constraintMultiplierTargetValue
ConstraintOffsetTarget	constraintOffsetTargetValue
ConstraintInsetTarget	constraintInsetTargetValue

RelatableTarget因为可以有好几种不同类型，因此无法在协议里声明一个通用的属性，在使用时进行类型转换判断：

```
public class ConstraintMakerRelatable {

    internal func relatedTo(_ other: ConstraintRelatableTarget, relation:
ConstraintRelation, file: String, line: UInt) -> ConstraintMakerEditable {
        let related: ConstraintItem
        let constant: ConstraintConstantTarget

        if let other = other as? ConstraintItem {
            //...
            related = other
            constant = 0.0
        } else if let other = other as? ConstraintView {
            related = ConstraintItem(target: other, attributes:
ConstraintAttributes.none)
            constant = 0.0
        } else if let other = other as? ConstraintConstantTarget {
            related = ConstraintItem(target: nil, attributes:
ConstraintAttributes.none)
            constant = other
        } else if #available(iOS 9.0, OSX 10.11, *), let other = other as?
ConstraintLayoutGuide {
            related = ConstraintItem(target: other, attributes:
ConstraintAttributes.none)
            constant = 0.0
        } else {
            fatalError("Invalid constraint. (\(file), \(line))")
        }

        //...
    }
}
```

Constraint

`Constraint`是Snapkit中表示约束的对象。

基础属性

最后`Constraint`还是要生成`LayoutConstraint`，所以基础属性都一致：

```
public final class Constraint {

    internal let sourceLocation: (String, UInt)
    internal let label: String?

    private let from: ConstraintItem
    private let to: ConstraintItem
    private let relation: ConstraintRelation
    private let multiplier: ConstraintMultiplierTarget
    private var constant: ConstraintConstantTarget
    private var priority: ConstraintPriorityTarget

    //...
}
```

增加了一个`sourceLocation`属性，用了记录生成这条约束的源代码位置，是一个`Tuple`，第一个值表示文件名，第二个值表示代码行数。这个属性可以在调试的时候输出相关信息。

layoutConstraints

在初始化时生成对应的`LayoutConstraint`数组。因为在Snapkit中的一个约束的属性是一个组合，所以可以对应几条原生约束。下面的代码在对应属性赋值完成后，遍历`layoutFromAttributes`生成对应的`LayoutConstraint`，然后添加到`layoutConstraints`中。

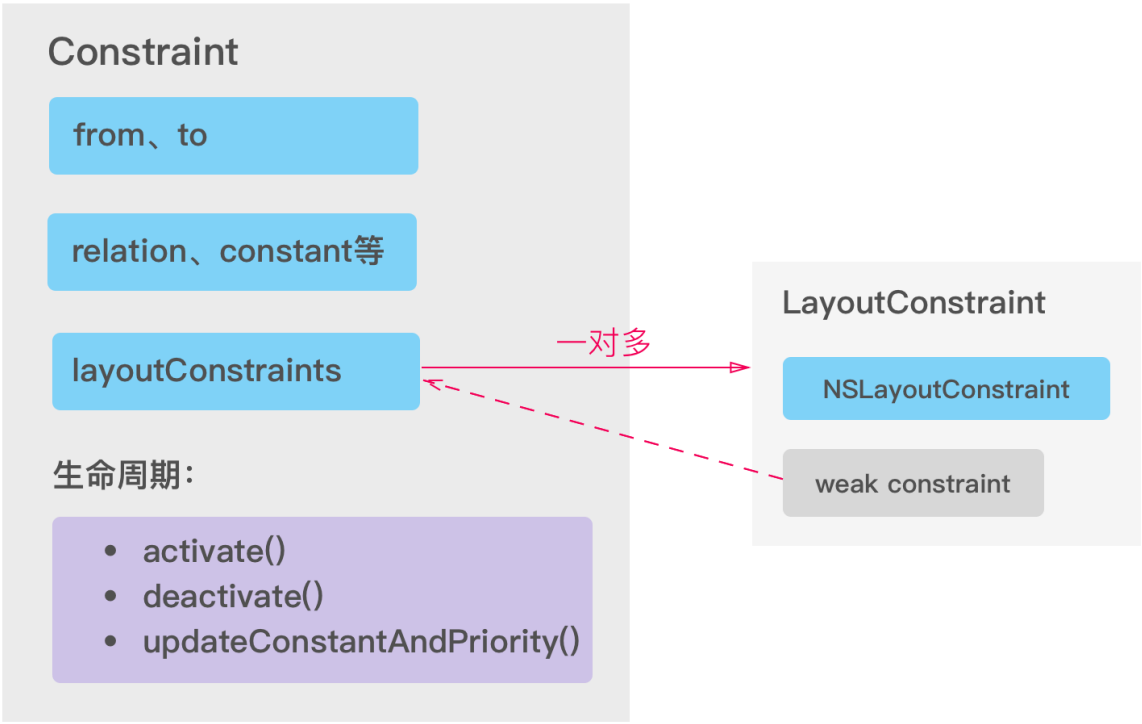
```
public var layoutConstraints: [LayoutConstraint]

    // MARK: Initialization

    internal init(from: ConstraintItem,
                  to: ConstraintItem,
                  relation: ConstraintRelation,
                  sourceLocation: (String, UInt),
                  label: String?,
                  multiplier: ConstraintMultiplierTarget,
                  constant: ConstraintConstantTarget,
                  priority: ConstraintPriorityTarget) {

        self.from = from
        self.to = to
        self.relation = relation
        self.sourceLocation = sourceLocation
        self.label = label
        self.multiplier = multiplier
        self.constant = constant
        self.priority = priority
        self.layoutConstraints = []

        //...
        let layoutFromAttributes = self.from.attributes.layoutAttributes
        for layoutFromAttribute in layoutFromAttributes {
           //...
           // 生成对应的LayoutConstraint
           let layoutConstraint = LayoutConstraint(
                item: layoutFrom,
                attribute: layoutFromAttribute,
                relatedBy: layoutRelation,
                toItem: layoutTo,
                attribute: layoutToAttribute,
                multiplier: self.multiplier.constraintMultiplierTargetValue,
                constant: layoutConstant
            )
            layoutConstraint.constraint = self
            self.layoutConstraints.append(layoutConstraint)
        }

}
```

管理约束生命周期

`LayoutConstraint`通过`isActivate`让这条约束生效。在`Constraint`也有类似的管理生命周期的函数：

```
public final class Constraint {
    //...
    func activateIfNeeded(updatingExisting: Bool = false) {
        //...
    }

    func deactivateIfNeeded() {
         //...
    }
}
```

图示



ConstraintDescription

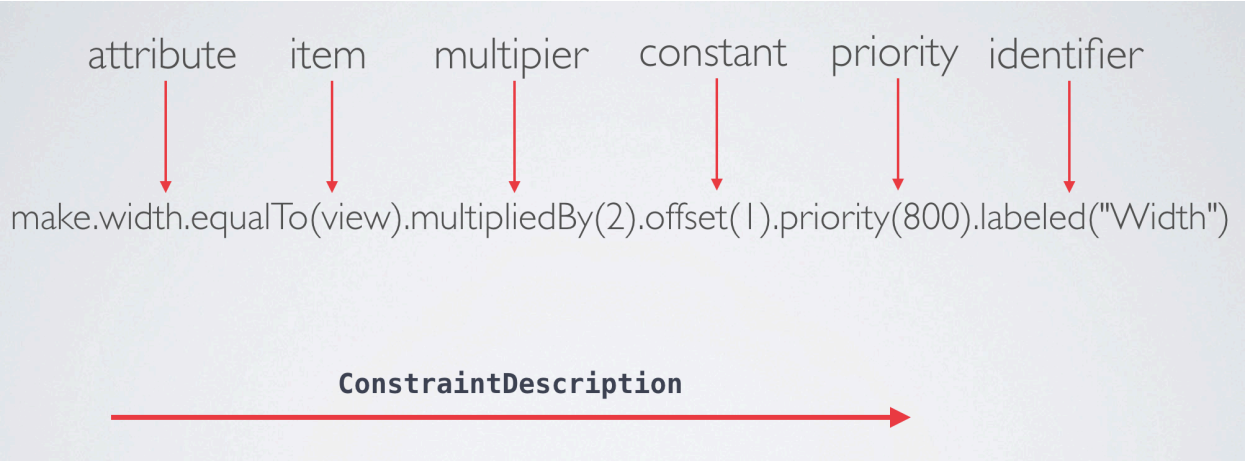
那么约束的一些默认值（multipier默认为1，constan默认为0）是在什么地方设置的呢？答案就是`ConstraintDescription`。

同时注意一下`Constraint`的基本属性的修饰符是`let`。Snapkit希望`Constraint`只有一个职责，就是生成原生约束，管理这些约束的生命周期。而对于这个约束的参数的不参与采集。这就是`ConstraintDescription`的意义：填充完整一个约束所需要的参数。

```
public class ConstraintDescription {

    internal let item: LayoutConstraintItem
    internal var attributes: ConstraintAttributes
    internal var relation: ConstraintRelation? = nil
    internal var sourceLocation: (String, UInt)? = nil
    internal var label: String? = nil
    internal var related: ConstraintItem? = nil
    internal var multiplier: ConstraintMultiplierTarget = 1.0
    internal var constant: ConstraintConstantTarget = 0.0
    internal var priority: ConstraintPriorityTarget = 1000.0

    internal lazy var constraint: Constraint? = {
        guard let relation = self.relation,
              let related = self.related,
              let sourceLocation = self.sourceLocation else {
            return nil
        }
        let from = ConstraintItem(target: self.item, attributes:
self.attributes)

        return Constraint(
            from: from,
            to: related,
            relation: relation,
            sourceLocation: sourceLocation,
            label: self.label,
            multiplier: self.multiplier,
            constant: self.constant,
            priority: self.priority
        )
    }()

    // MARK: Initialization

    internal init(item: LayoutConstraintItem, attributes:
ConstraintAttributes) {
        self.item = item
        self.attributes = attributes
    }

}
```

`ConstraintDescription`定义了和`Constraint`一样的基本参数属性，只是属性是可变的`var`。通过`constraint`属性生成`Constraint`类型的实例。也很清楚的看到在这里设置了multiplier、constant、priority的默认值。

什么时候填充的这些值呢?



使用make后的每一步里设置的值最后都会设置到一个`ConstraintDescription`实例中。最后从这个实例的`constraint`属性生成一个真正的`Constraint`。

ConstraintViewDSL

现在从使用Snapkit的代码一步步深入来看。

```
blueView.snp.makeConstraints { (make) in
            make.width.height.equalTo(100)
            }
```

snp

```
public extension ConstraintView {
    //...
    public var snp: ConstraintViewDSL {
        return ConstraintViewDSL(view: self)
    }

}
```

snp是UIView的一个扩展属性，每次返回一个新的`ConstraintViewDSL`实例，这个实例的初始化方法会保存这个view。

contentHuggingPriority、contentCompressionResistancePriority

在DSL中封装了设置HuggingPriority和CompressionResistancePriority的属性。在set的时候调用UIView的方法。

```
    public var target: AnyObject? {
        return self.view
    }

    internal let view: ConstraintView

    internal init(view: ConstraintView) {
        self.view = view


    public var contentHuggingHorizontalPriority: Float {
        get {
            return self.view.contentHuggingPriority(for: .horizontal)
        }
        set {
            self.view.setContentHuggingPriority(newValue, for: .horizontal)
        }
    }

    public var contentHuggingVerticalPriority: Float {
        //...
    }

    public var contentCompressionResistanceHorizontalPriority: Float{
        //...
    }

    public var contentCompressionResistanceVerticalPriority: Float {
        //...
    }
```

ConstraintAttributesDSL

Snapkit中设置约束的时候有时还会这么写：

```
make.left.equalTo(self.view.snp.right)
```

这里的`snp.right`属性就是定义在`ConstraintAttributesDSL`里。`ConstraintViewDSL`实现了这个`ConstraintAttributesDSL`协议。

```
public struct ConstraintViewDSL: ConstraintAttributesDSL {

   public var target: AnyObject? {
        return self.view
    }
 }

 extension ConstraintBasicAttributesDSL {

    public var left: ConstraintItem {
        return ConstraintItem(target: self.target, attributes:
ConstraintAttributes.left)
    }

    public var top: ConstraintItem {
        return ConstraintItem(target: self.target, attributes:
ConstraintAttributes.top)
    }

    //...
 }
```

每次都创建一个`ConstraintItem`对象。这个对象前面已经介绍过，包含target和ConstraintAttributes，表示一个item。

makeConstraints

这里当然还少不了`makeConstraints`函数。同时还定义了updateConstraints、remakeConstraints、removeConstraints等处理约束的方法。

很容易看出这里面操作的核心就是`ConstraintMaker`。

```
    public func prepareConstraints(_ closure: (_ make: ConstraintMaker) ->
Void) -> [Constraint] {
        return ConstraintMaker.prepareConstraints(item: self.view, closure:
closure)
    }

    public func makeConstraints(_ closure: (_ make: ConstraintMaker) ->
Void) {
        ConstraintMaker.makeConstraints(item: self.view, closure: closure)
    }

    public func remakeConstraints(_ closure: (_ make: ConstraintMaker) ->
Void) {
        ConstraintMaker.remakeConstraints(item: self.view, closure: closure)
    }

    public func updateConstraints(_ closure: (_ make: ConstraintMaker) ->
Void) {
        ConstraintMaker.updateConstraints(item: self.view, closure: closure)
    }

    public func removeConstraints() {
        ConstraintMaker.removeConstraints(item: self.view)
    }
```

ConstraintMaker

先来看Maker使用的情况：

```
make.width.equalTo(100)
```

make的作用是通过各种函数把`ConstraintDescription`的值填充完整。

```
public class ConstraintMaker {

    public var left: ConstraintMakerExtendable {
        return self.makeExtendableWithAttributes(.left)
    }

    private let item: LayoutConstraintItem
    private var descriptions = [ConstraintDescription]()

    internal init(item: LayoutConstraintItem) {
        self.item = item
        self.item.prepare()
    }

    internal func makeExtendableWithAttributes(_ attributes:
ConstraintAttributes) -> ConstraintMakerExtendable {
        let description = ConstraintDescription(item: self.item, attributes:
attributes)
        self.descriptions.append(description)
        return ConstraintMakerExtendable(description)
    }

    //...
}
```

maker初始化时接收一个item作为参数，在这里调用了item的`prepare()`函数，所以每个UIView如果要被设置约束都会在这里设置`translatesAutoresizingMaskIntoConstraints`：

```
func prepare() {
       if let view = self as? ConstraintView {
           view.translatesAutoresizingMaskIntoConstraints = false
       }
}
```

接着创建了一个空的`ConstraintDescription`数组。

每次设置maker的约束属性比如`make.left`、`make.center`就会创建一个对应的`ConstraintDescription`添加到`descriptions`这数组中。

ConstraintMakerExtendable

`ConstraintMakerExtendable`继承自`ConstraintMakerRelatable`。初始化方法定义在`ConstraintMakerRelatable`中：

```
public class ConstraintMakerRelatable {

    internal let description: ConstraintDescription

    internal init(_ description: ConstraintDescription) {
        self.description = description
    }
    //...
}
```

与此类似`ConstraintMakerEditable`继承`ConstraintMakerPriortizable`，`ConstraintMakerPriortizable`继承`ConstraintMakerFinalizable`，`ConstraintMakerFinalizable`也定义了一样的初始化方法：

```
public class ConstraintMakerFinalizable {

    internal let description: ConstraintDescription

    internal init(_ description: ConstraintDescription) {
        self.description = description
    }

    @discardableResult
    public func labeled(_ label: String) -> ConstraintMakerFinalizable {
        self.description.label = label
        return self
    }

    public var constraint: Constraint {
        return self.description.constraint!
    }

}
```

make.left

仔细看`left`的源码，会发现返回的是`self`：

```
    public var left: ConstraintMakerExtendable {
        self.description.attributes += .left
        return self
    }

    public var top: ConstraintMakerExtendable {
        self.description.attributes += .top
        return self
    }
```

make在第一层设置的都是attributes，设置的attributes由description保存。

ConstraintMakerRelatable

在确定了属性后就要确定item2， 第二层ConstraintMakerRelatable定义了`equalto`函数：

```
    @discardableResult
    public func equalTo(_ other: ConstraintRelatableTarget, _ file: String =
#file, _ line: UInt = #line) -> ConstraintMakerEditable {
        return self.relatedTo(other, relation: .equal, file: file, line:
line)
    }

    @discardableResult
    public func lessThanOrEqualTo(_ other: ConstraintRelatableTarget, _
file: String = #file, _ line: UInt = #line) -> ConstraintMakerEditable {
        return self.relatedTo(other, relation: .lessThanOrEqual, file: file,
line: line)
    }

     internal func relatedTo(_ other: ConstraintRelatableTarget, relation:
ConstraintRelation, file: String, line: UInt) -> ConstraintMakerEditable {
        let related: ConstraintItem
        let constant: ConstraintConstantTarget
         // 对related（toItem）、constant进行赋值
        if let other = other as? ConstraintItem {
            //...
            related = other
            constant = 0.0
        } else if let other = other as? ConstraintView {
            related = ConstraintItem(target: other, attributes:
ConstraintAttributes.none)
            constant = 0.0
        } else if let other = other as? ConstraintConstantTarget {
            related = ConstraintItem(target: nil, attributes:
ConstraintAttributes.none)
            constant = other
        } else if #available(iOS 9.0, OSX 10.11, *), let other = other as?
ConstraintLayoutGuide {
            related = ConstraintItem(target: other, attributes:
ConstraintAttributes.none)
            constant = 0.0
        } else {
            fatalError("Invalid constraint. (\(file), \(line))")
        }
        // 生成 ConstraintMakerEditable 返回，进入下一个填充值的流程中
        let editable = ConstraintMakerEditable(self.description)
        editable.description.sourceLocation = (file, line)
        editable.description.relation = relation
        editable.description.related = related
        editable.description.constant = constant
        return editable
    }
    //...
```

通过源码可以看出，调用`equalto`的时候记录了源码的位置，填充了related（toItem）、constant。在这里已经确认了from、to、relation、constant。把description继承送给ConstraintMakerEditable。

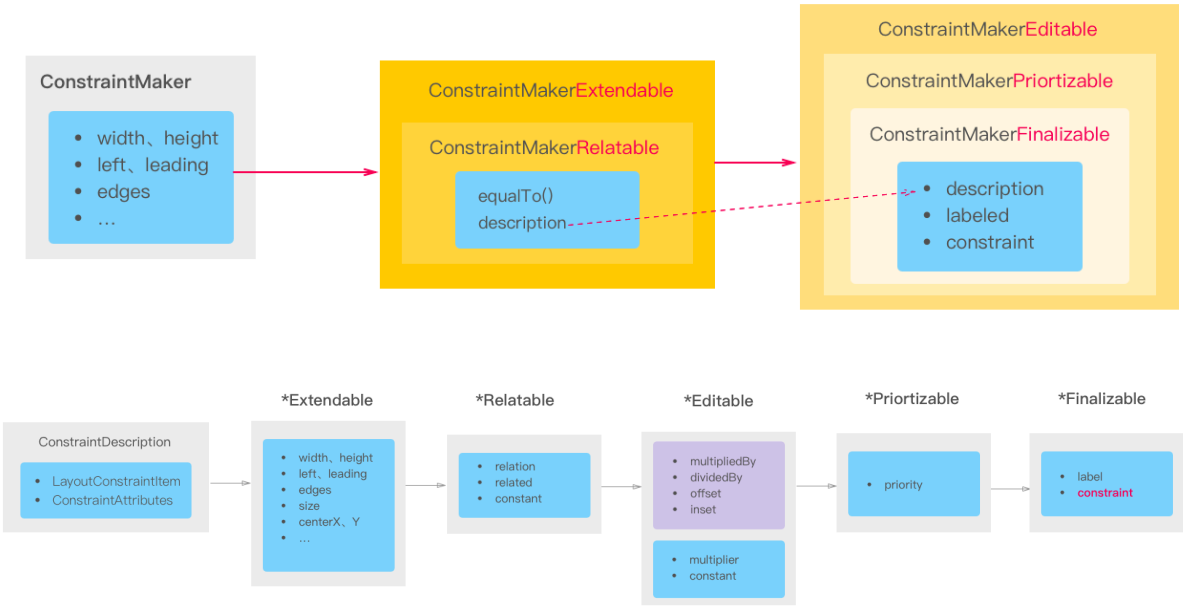
MakerEditable、MakerPriortizable、MakerFinalizable

接着的步骤就和上面的一样，每步的属性设置完成以后，设置父类的对应属性。

ConstraintMakerEditable	ConstraintMakerPriortizable	ConstraintMakerFinalizable
multipliedBy	priority	labeled
dividedBy		
offset		
insets		

最后从`ConstraintMakerFinalizable`保存的description里取出约束。

图示



makeConstraints

真正创建约束的地方在Maker里：

```
    internal static func makeConstraints(item: LayoutConstraintItem,
closure: (_ make: ConstraintMaker) -> Void) {
        let maker = ConstraintMaker(item: item)
        closure(maker)
        var constraints: [Constraint] = []
        for description in maker.descriptions {
            guard let constraint = description.constraint else {
                continue
            }
            constraints.append(constraint)
        }
        for constraint in constraints {
            constraint.activateIfNeeded(updatingExisting: false)
        }
    }
```

流程到这里已经非常清晰，显示传入当前的item，创建一个Maker。接着填充完这个maker里的`ConstraintDescription`数组，接着从这个数组中生成对应的约束，最后激活这些约束。

代码总结

Snapkit完整灵活的提供了Auto Layout的能力。对外的API也做的非常简洁。使用了协议扩展和继承来复用代码，很好的控制了代码的复杂度，最大的一个类代码行数没有超过300行。

当我们读源码的时候在读什么?

我想除了了解它的实现原理外，还应该注意到它的结构设计。当一个开源库足够流行，可以说明它的解决方案得到了开发者的认可，在实现目的和代码实现上达到了一个良好的平衡。我们也应该思考学习它解决问题的模式和架构，尝试理解它做出的选择与取舍。

参考链接

Auto Layout Guide

Visual Format Language

Snapkit

DEMO