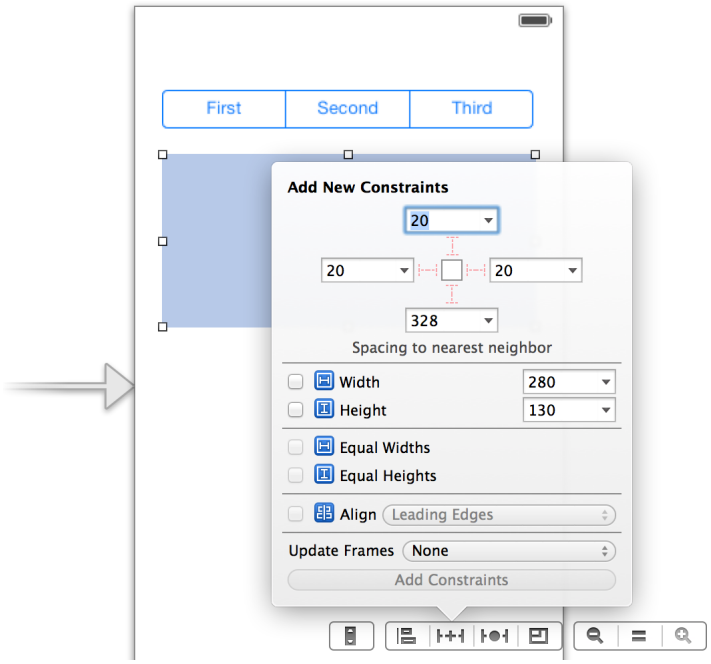
Auto Layout ガイド

Developer

目次

概要

Auto Layoutはアプリケーションのユーザインターフェイス（特に画面レイアウト）を作成するシステムで、要素間の関係を数学的に記述することによりこれを行います。この関係を、個々の要素に対する制約、あるいは要素間の制約として定義します。Auto Layoutを導入すれば、画面の大きさや向き、地域設定（言語など）の違いに応じて動的に変化する、汎用的な画面レイアウトを作り出すことができます。



Auto LayoutはXcode 5のInterface Builderに組み込まれており、iOS用、OS X用のどちらのアプリケーションにも適用できます。プロジェクトを新規に作成すると、初めからこの機能が有効になっています。Auto Layoutが有効になっていない既存のプロジェクトを扱う場合は、“Auto Layoutの導入”（40 ページ）を参照してください。

ユーザインターフェイスを開発する典型的な手順は、Interface Builder上で、ビューやコントロール部品を作り、位置や大きさを調整し、さまざまにカスタマイズすることから始まります。位置その他の設定が決まった後、Auto Layoutの制約を追加して、向きや大きさ、地域設定の違いに応じ、インターフェイスが動的に変わるようにしていきます。

はじめに

Auto Layout（Xcode 5に付属）は、制約(Constraint)に基づいて画面レイアウトを調整する仕組みです。OS X用、iOS用のアプリケーションに適用するレイアウトを、短時間で作成し、容易に保守することができます。次のような特徴があります。

- 必要に応じていつでも制約を追加できる
- Ctrlキーを押しながらドラッグする操作、あるいはメニューコマンドにより、簡単に制約を追加できる
- 制約とフレームをそれぞれ独立に更新できる
- 動的に配置が決まるビューに、プレイスホルダ制約を指定できる
- 相反する制約、配置がひと通りに決まらないビューなどの問題は、状況を視覚的に確認しながら解消できる

この資料の構成

Auto Layoutの使い方について、次の各章に分けて解説します。

- "Auto Layoutの考え方"（6 ページ）では、Auto Layoutの導入に当たって知っておくべき、重要な考え方を示します。
- "Interface Builderにおける制約の操作"（9 ページ）では、Interface Builderを使って制約を定義、編集する手順を説明します。
- "Auto Layoutを操作するプログラム"（14 ページ）では、Auto Layoutを操作するプログラムの書き方を説明します。
- "配置に関する問題の解決"（17 ページ）では、レイアウトが想定どおりにならないとき、原因を特定し、修正する方法を検討します。
- "Auto Layoutの使用例"（24 ページ）ではAuto Layoutを活用してアプリケーションを開発する典型的な例を示します。
- "Auto Layoutと連携して動作するカスタムビューの実装"（37 ページ）では、Auto Layoutの機構とやり取りする、カスタムビューの実装法を示します。
- "Auto Layoutの導入"（40 ページ）では、Auto Layoutを採り入れていない既存のプロジェクトに、後から導入する手順を解説します。
- "視覚的書式言語"（42 ページ）では、制約をプログラムとして記述する言語の仕様を説明します。

Auto Layoutの考え方

Auto Layoutの基盤を成すのは**制約 (Constraint)** という考え方です。これはインターフェイスを構成する要素の配置（レイアウト）を決める規則です。ある要素の幅を指定する制約、別の要素からの水平距離を指定する制約などがあります。こういった制約を追加、削除し、あるいはその設定（プロパティ）を変更することにより、実際に画面に現れる配置が変わります。

実行時に要素の位置を計算する際、Auto Layoutの機構はあらゆる制約を同時に考慮し、できるだけすべての制約を満たすように位置を決めようとします。

制約の基本事項

制約とは、人が聞いて分かるように配置を表した言明を、数学的に表現したものと考えることができます。たとえばボタンの位置を定義する場合、「このボタンを収容するビューの左辺から、ボタンの左端を20ポイント離す」などと言明するでしょう。これを形式的に言い換えると「`button.left = (container.left + 20)`」となります。一般にこのような式は、「`y = m*x + b`」という形をしています。ただし：

- `y`と`x`はビューの属性です。
- `m`と`b`は浮動小数点数です。

属性は、`left`、`right`、`top`、`bottom`、`leading`、`trailing`、`width`、`height`、`centerX`、`centerY`、`baseline`のいずれかです。

属性`leading`と`trailing`は、英語など左から右へ書く言語の場合は`left`および`right`と同じですが、ヘブライ語やアラビア語のように右から左へ書く言語ならば逆に、`leading`および`trailing`と`right`および`left`と同じ意味になります。制約の定義には、`leading`および`trailing`を使うのが原則です。通常は、どの言語でも適切に配置されるよう、`leading`と`trailing`を使ってください。ただし、（分割ビューにおけるマスタペインと詳細ペインの関係のように、）言語によらず左右が同じであるべき制約の場合を除きます。

制約には次のようなプロパティがあります。

- **定数値 (Constant value)** ポイント単位の物理的な大きさまたはオフセット。

- **関係(Relation)** ビューの属性値として、定数値以外に関係式や不等式（≧など）も指定できます。たとえば幅について「`width >= 20`」という指定、あるいは要素間に「`textview.leading >= (superview.leading + 20)`」といった指定ができるのです。
- **優先度 (Priority level)** 制約には優先度を指定できます。優先度の高い制約をまず満たした後で、優先度の低い制約を考慮するようになっています。デフォルトの優先度は「必須」（`NSLayoutPriorityRequired`）というもので、制約を完全に満たさなければならないことを表します。一方、「必須」優先度でない制約については、完全ではないにしても、できるだけそれに近づけようとします。

  優先度をうまく指定すれば、条件つきの制約を表現できます。たとえばあるコントロール部品について、通常はその内容に応じた大きさにするが、より重要な要因がある場合はその限りでない、という形の制約を設定できるのです。優先度について詳しくは、`NSLayoutPriority`を参照してください。

制約が複数あればすべて同時に適用され、一方がもう一方を打ち消すことはありません。制約があるところに、同じ種類の別の制約をさらに適用しても、元の制約が無効になるわけではないのです。たとえばあるビューに対して、幅の制約を重ねて設定したとしても、元からあった制約は無効になりません。必要ならば明示的に削除する必要があります。

若干の制限はありますが、ビュー階層をまたがる制約も可能です。たとえばOS X用の「メール(Mail)」アプリケーションの場合、ツールバーの「Delete」ボタンはメッセージテーブルと揃うように配置されています。「Desktop Preferences」では、ウインドウ下部のチェックボックスが、分割ビューペインの境界に揃っています。

ただし、ビュー階層をまたがる制約を設定できない場合があります。たとえば、ビュー階層内に、サブビューのフレームを独自の方法で設定するビュー（`UIView`の`layoutSubviews`メソッド（または`NSView`の`layout`メソッド）を独自に実装）がある、という状況です。また、（スクロールビューのように）座標変換機能があるビューも同様です。こういったビューは、レイアウト上の「障壁」になっていると考えるとよいでしょう。内側と外側の世界に分かれていて、制約で結びつけることはできないのです。

固有の寸法

ボタンなど、リーフレベルのビューは通常、あるべきサイズに関して、位置決めをするコードよりもよく知っています。その際に手がかりとして使うのが、**固有の寸法 (Intrinsic Content Size)**と呼ばれるものです。ビューには何らかの「コンテンツ」が含まれている場合に、その寸法をレイアウトの機構に伝えるのです。

テキストラベルなどの要素には、通常、この固有の寸法を使うよう設定するとよいでしょう（「Editor」＞「Size To Fit Content」コマンド）。こうしておけば、言語によって内容が変わったとき、それに応じて適切に大きさが変わります。

アプリケーションアーキテクチャ

Auto Layoutアーキテクチャでは、レイアウト処理をコントローラとビューとで役割分担しています。配置に関するあらゆる情報をコントローラ側に集約し、ビューの配置を計算する代わりに、ビューの側で自分自身の配置を制御するようになっているのです。この方針により、コントローラ側のロジックが複雑になるのを緩和できます。また、レイアウトに関係するコードを修正することなく、容易にビューを再設計できるようになります。

それでも、実行時に制約を追加し、削除し、調整するコントローラオブジェクトが欲しくなるかも知れません。制約を操作するコードの書き方については、“Auto Layoutを操作するプログラム”（14ページ）を参照してください。

コントローラの役割

ビューには固有の寸法を指定しますが、さらにその重要度も指定可能です。たとえばボタンの寸法は、デフォルトでは次のようになっています。

- 縦方向には、コンテンツを内包する自然な高さ（重要度：高）。
- 横方向には、コンテンツを内包する幅だが、溝縁部分とラベルとの間に余白があっても可（重要度：低）。
- コンテンツの圧縮（縮小や切り抜き）は縦横とも不可（重要度：高）。

たとえば隣り合う2つのボタンがあり、拡大する余地が生じた場合に、どちらを大きくするか決めるのはコントローラの役割です。一方のみを拡大しても、両方を同じだけ拡大しても構いません。あるいは、元の大きさに比例して拡大することも考えられます。逆に、中身を縮めるか一部を切り落とさなければ、両方のボタンは収容できない、という場合、まず一方だけにこの処理を施す方法や、どちらにも同じように適用する方法が考えられるでしょう。この判断もコントローラの役割です。

`UIView`のインスタンスについて、中身を密着させ、あるいは縮める処理を施す優先度は、`setContentHuggingPriority:forAxis:`および`setContentCompressionResistancePriority:forAxis:`で設定します（`NSView`の場合は`setContentHuggingPriority:forOrientation:`および`setContentCompressionResistancePriority:forAxis:`）。UIKitやAppKitに付属のビューは、デフォルトではいずれも、`NSLayoutPriorityDefaultHigh`または`NSLayoutPriorityDefaultLow`という値が与えられています。

Interface Builderにおける制約の操作

制約の追加、編集、削除には、Interface Builderのレイアウトツールが、画面上で状況を見ながら操作できて便利です。たとえばControlキーを押しながら2つのビュー間をドラッグするだけで、容易に制約を追加できます。さまざまなポップアップウインドウを活用すれば、いくつもの制約を一括して作成するような操作も簡単です。

制約を追加する

オブジェクトライブラリにある要素を摑んでドラッグし、Interface Builderのキャンバス上にドロップした後、この要素をドラッグすることにより、制約なしの状態で自由に配置を決めることができます。このまま制約を与えずにビルド、実行した場合、要素の幅や高さは一定で、スーパービューの左上隅を基準とする所定の位置に固定です。ウインドウの大きさを変えても、要素の位置や大きさは変わりません。大きさや向きの変化に適切に応じるためには、制約を追加する必要があります。

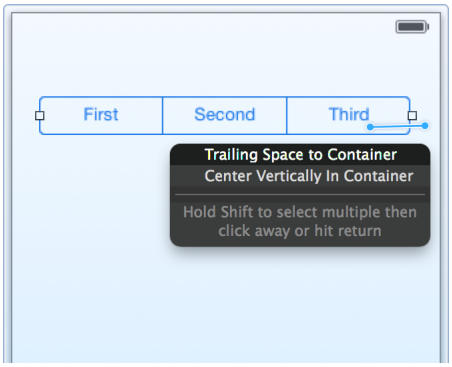
> **Important:** 制約なしでユーザインターフェイスを構築しても、Xcodeに警告やエラーが現れることはありませんが、そのままでアプリケーションを出荷することは避けてください。

制約を追加する手段は、そのきめ細かさの程度や、一括して追加する個数に応じていくつかあります。

Control-ドラッグ操作で制約を追加する

最も簡単なのは、Controlキーを押した状態で、キャンバス上のビューからドラッグする方法です。アウトレットやアクションにリンクを生成する手順とよく似ています。制約の種類や適用対象が明確に分かっているならば、制約をひとつ作成する手段としては最も簡単で、きめ細かく指定できる方法です。

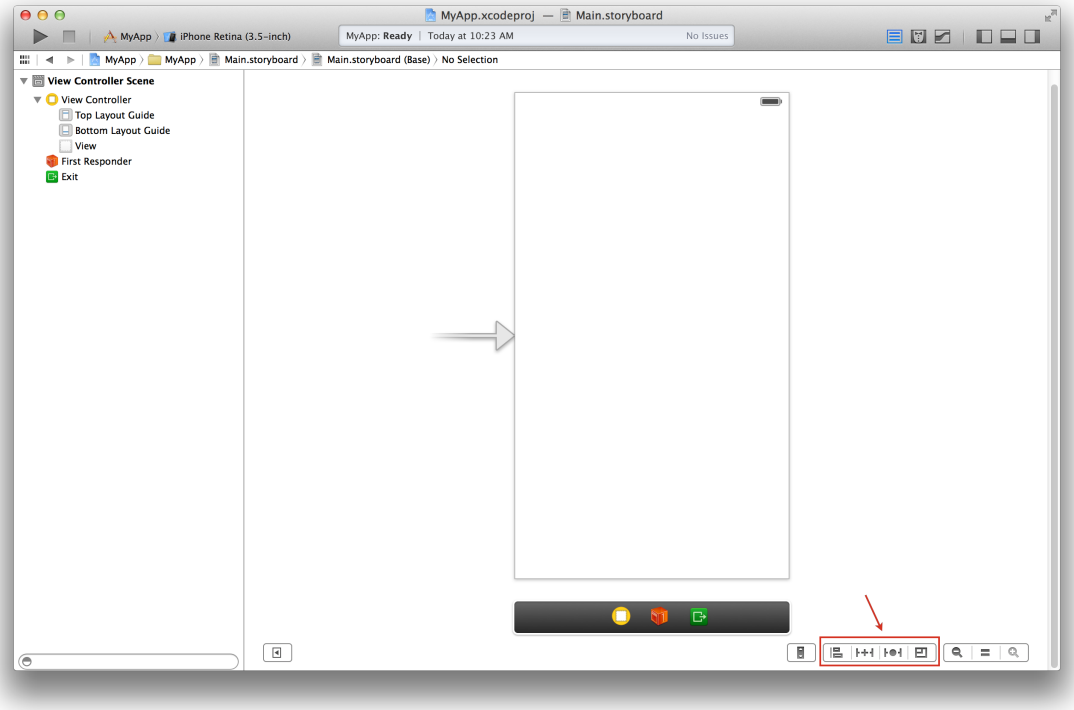
要素からその要素自身に、あるいはこれを収容するコンテナやその他の要素に向かってドラッグします。ドラッグの行き先と方向によって、Auto Layoutは追加する制約を推測し、選択肢をしぼり込むようになっています。たとえば、ある要素から出発し、右方向に進んでコンテナまでドラッグした場合、要素の後側に空ける間隔（trailing space）を固定する、またはコンテナの高さ方向に対して中央寄せ（center vertically）にする、という選択肢が現れます。



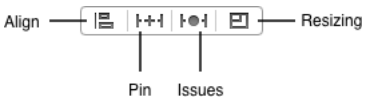
Tip: このとき現れるメニューから、複数の制約をまとめて選択したい場合は、CommandキーまたはShiftキーを押しながら操作してください。

「Align」メニュー、「Pin」メニューで制約を追加する

Interface Builderのキャンバス上にある、「**Auto Layout**」**メニュー**で制約を追加することも可能です。

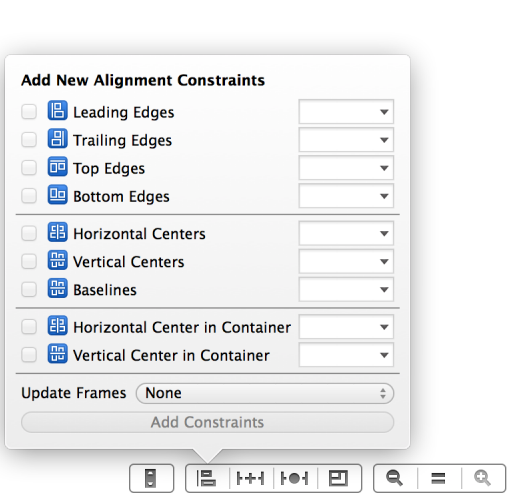


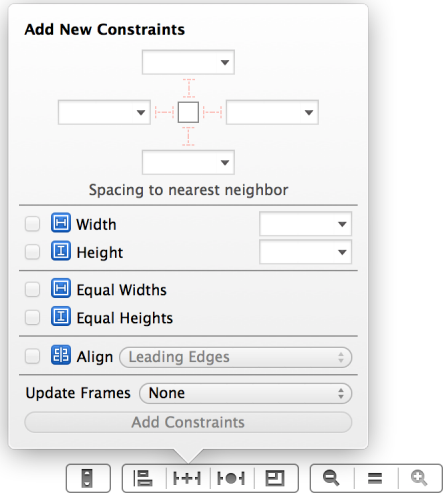
揃えや空間的配置に関する制約を追加する以外にも、配置上の問題を解決し、あるいは大きさが変わる動作を決めるコマンドがあります。



- **Align**。揃えに関する制約を追加します。コンテナの中央に寄せる、2つのビューの左辺に揃える、などが可能です。
- **PIN**。空間的配置に関する制約を追加します。ビューの高さを定義する、別のビューからの水平距離を指定する、などが可能です。
- **Issues**。配置上の問題を解決するためのコマンドで、どのような制約を追加/削除すればよいか、候補が現れます（“配置に関する問題の解決” （17 ページ）を参照）。

- **Resizing**。大きさが変わったとき、制約にどのような影響を及ぼすか、を指定します。



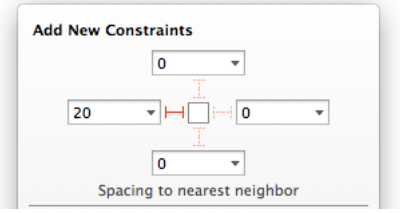


要素をひとつだけ選択している場合、複数の要素を要する制約の選択肢は無効になります。

To :「Align」メニュー、「Pin」メニューで制約を追加するには

1. 該当する制約のチェックボックスをオンにしてください。

   「Spacing to nearest neighbor」制約については、要素のどちら側に関する制約か、該当する方向の赤線を選択します。



   隣接しないビューとの関係で制約を追加する場合は、該当するビューを選択しなければなりません。値の入力欄には黒い三角マークがついていますが、これをクリックすると、ビューを選択するドロップダウンメニューが現れます。

2. 制約に応じた定数値を入力します。
3. ボタンを押して制約を追加します。
   - 「Add Constraints」ボタンは、選択した要素に、新たに制約を追加するために使います。
   - 「Add and Update Frames」ボタンは、同様に制約を追加した後、すべての制約をできるだけ満たすよう、要素の位置を調整するために使います。

> **注意:** この2つのボタンを押すと、その都度まったく新たに制約が追加されます。既存の制約を上書きするのではありません。定義済みの制約を編集する手順については、"制約を編集する"（13 ページ）を参照してください。

不足している制約、候補として示された制約を追加する

「Issues」メニューでは、設計作業の出発点として適切な制約を追加できます。多数の変更を手早く行いたい場合にも有用です。

多数の制約が必要で、ひとつひとつ追加していくのが面倒であれば、「Issues」＞「Add Missing Constraints」コマンドが役に立つかも知れません。現在の配置の状態から必要な制約を推測し、自動的に追加するようになっています。

一連の制約を取り消して元に戻したい、あるいはまったく最初からやり直したい場合は、「Issues」＞「Reset to Suggested Constraints」コマンドでいったん削除し、適切な制約のみ追加した状態にするとよいでしょう。これは「Clear Constraints」コマンドと「Add Missing Constraints」コマンドの組と同等です。

制約を編集する

制約のプロパティ（定数値、関係、優先度）を変更できます。これは、キャンバス上で制約をダブルクリックして値を編集するか、または制約を選択し、「Attributes」インスペクタを開いて行います。ただし制約の種類は変更できません（たとえば幅の制約を高さの制約に変更することは不可）。

制約を削除する

制約の削除はいつでも、キャンバス上あるいはアウトラインビュー上で選択し、「Delete」キーを押すことにより行えます。

Auto Layoutを操作するプログラム

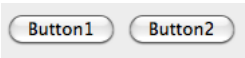
Auto LayoutはInterface Builder上で、目で見て確認しながら操作すると便利ですが、制約を作成、追加、削除、調整するコードを記述することも可能です。たとえば、実行時にビューを追加/削除する場合、制約もプログラムで（実行時に）追加して、大きさや向きの変化に適切に応じられるようにしなければなりません。

制約をプログラムで作成する

プログラム上、制約は`NSLayoutConstraint`のインスタンスで表します。制約の生成には通常、`constraintsWithVisualFormat:options:metrics:views:`メソッドを使います。

第1引数には**視覚的書式文字列(visual format string)**を指定します。レイアウト上の制約を視覚的に表した文字列です。この**視覚的書式文字列**は、読みやすさを重視して考えられています。ビューは角括弧で囲み、ビュー間の接続はハイフンで表します（2個のハイフンの間に数字をはさむと、ビュー間の間隔をポイント数で指定したことになります）。この書式文字列の例と文法を“視覚的書式言語” （42 ページ）で説明します。

例として、2つのボタン間の制約を表してみましょう。



たとえば次のように表します。

```
[button1]-12-[button2]
```

ハイフンが1つだけであればAqua標準の間隔を表すので、次のように記述するだけでも構いません。

```
[button1]-[button2]
```

ビュー名は`views`辞書から取得します。書式文字列に記述したビュー名をキー、対応するビューオブジェクトを値とする辞書です。この辞書を生成するには`NSDictionaryOfVariableBindings`を使うと便利でしょう。辞書のキーは、このマクロに渡した値に対応する変数名と同じになります。制約を生成するコードは次のようになります。

```
NSDictionary *viewsDictionary =
                NSDictionaryOfVariableBindings(self.button1, self.button2);
NSArray *constraints =
        [NSLayoutConstraint constraintsWithVisualFormat:@"[button1]-[button2]"
                                options:0 metrics:nil views:viewsDictionary];
```

視覚的書式言語は、表現力よりも見た目の分かりやすさを重視しています。実際のユーザインターフェイスで有用な制約はほとんど表現できますが、いくつか例外があります。有用だけれどもこの方法で表現できないものとしては、縦横比に関する制約（たとえば「`imageView.width = 2 * imageView.height`」）があります。このような制約を生成するためには`constraintWithItem:attribute:relatedBy:toItem:attribute:multiplier:constant:`メソッドを使います。

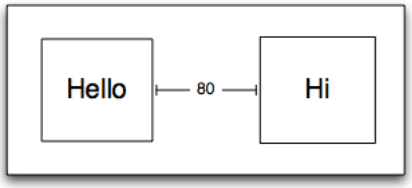
たとえば次のコードで、先に示した「`[button1]-[button2]`」と同じ制約を生成できます。

```
[NSLayoutConstraint constraintWithItem:self.button1 attribute:NSLayoutAttributeRight
                    relatedBy:NSLayoutRelationEqual toItem:self.button2
                attribute:NSLayoutAttributeLeft multiplier:1.0 constant:-12.0];
```

制約を組み込む

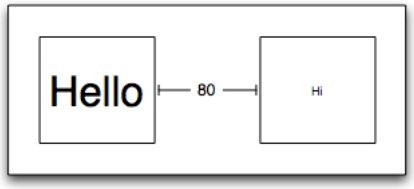
制約を実際に制約として働かせるためには、ビューに組み込まなければなりません。組み込み先のビューは、その制約が影響を及ぼす各ビューの祖先でなければならず、通常は、当該各ビューに共通の祖先のうち、直近のものを選びます（これは`NSView`のAPIにおける意味の*祖先*なので、ビューは自分自身の（0世代前の）祖先でもあります）。制約はこの祖先ビューの座標系に基づいて解釈します。

たとえば、「`[zoomableView1]-80-[zoomableView2]`」という制約を、`zoomableView1`と`zoomableView2`の共通の祖先であるコンテナビューに組み込んだとしましょう。

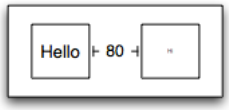


この80はコンテナの座標系における間隔であって、制約を描画すれば上図のようになります。

zoomableView1やzoomableView2の座標変換行列が変わっても、ビュー間の間隔は変わりません。



しかしコンテナ自身の座標変換行列が変われば、間隔も変わります。



`NSView`には、制約を追加する`addConstraint:`メソッド、既存の制約を削除する`removeConstraint:`メソッド、組み込まれている制約を調べる`constraints`メソッド、その他があります。`NSView`にはさらに、`fittingSize`というメソッドもあります。これは`NSControl`の`sizeToFit`メソッドに似ていますが、コントロール部品ではなく任意のビューを対象とします。

`fittingSize`メソッドは、当該ビューの観念的なサイズを返します。*レシーバのサブツリーに組み込まれている制約のみを考慮して、これを満たす最小のサイズを求めるの*です。とは言っても、一般的な意味でビューに「最適」なサイズではありません。制約ベースのシステムでは、（あらゆる制約を考慮し、）実現可能なサイズを求めなければならないからです。

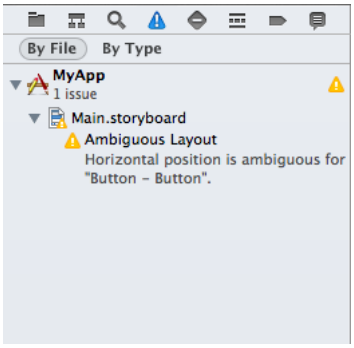
配置に関する問題の解決

Auto Layoutに関する問題としては、制約どうしの衝突、制約の不足、最終的な配置がひと通りに決まらない（不定な）制約などが考えられます。Auto Layoutはこのような場合、できるだけアプリケーションが問題なく動作するよう試みますが、開発の際、どうすれば問題があることを把握できるか理解しておくことが重要です。Auto Layoutには、原因を特定し解決するための機能がいくつかあります。Interface Builder上に、目に見えるようヒントを表示し、解決に役立つようにしているのもその現れです。

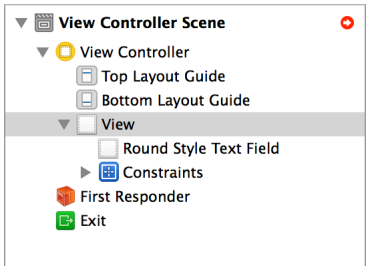
問題を特定する

Interface Builderは、Auto Layoutに関する問題点を各所に表示するようになっています。

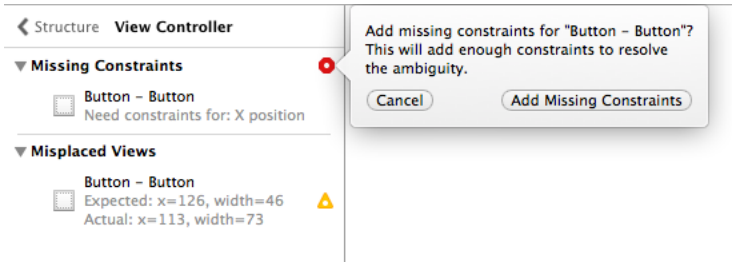
「Issues」ナビゲータ。問題点を種類ごとに表示します。



Interface Builderのアウトラインビュー。キャンバス上で編集している最上位レベルのビューに問題があれば、Interface Builderのアウトラインビューに、その旨を表す矢印が現れます。制約が衝突する、あるいは最終的な配置が不定である場合は赤、ビューが誤配置になる場合は黄色の矢印です。



この矢印を押すと問題点が、関係するコントロール部品や制約とともに、種類ごとに列挙されます。問題点の記述に添えられたエラー/警告シンボルを押すと、詳しい状況と対処法の候補が現れます。

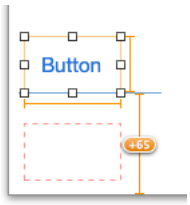


キャンバス上。誤配置になる、あるいは最終的な配置が不定である場合は橙、制約どうしが衝突している場合は赤になります。さらに、実行したとき実際に表示される位置を予測し、赤い点線の枠で示します。この問題については以降の節で説明します。

ビューの誤配置を解決する

Interface Builderでは、制約とフレームを別々に扱います。ビューについて、キャンバス上に見えているフレームと、制約に基づき実行時に決まる位置とが一致していない状況を、**ビューの誤配置 (view misplacement)**と言います。

この場合Interface Builderは、制約によって決まる（実際に満たされることはない）位置を橙色で示し、さらにその偏差も表示するようになっています。実行時に現れるビューの位置は、赤い点線の枠で示します。



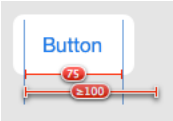
To :誤配置を解決するには

Do one of the following:

- 「Issues」＞「Update Frames」コマンドを用いる方法。要素の位置が、誤配置する前の状態に戻ります。フレームは他の制約を満たすように変わります。
- 「Issues」＞「Update Constraints」コマンドを用いる方法。（誤配置された）要素に応じて、制約の方を更新します。フレームの位置に合わせて制約が変化するのです。

制約間の衝突を解決する

制約間に**衝突(Conflict)**が起こっているとは、同じ要素に対して異なる幅を指定しているなど、同時に満たすことができない状況を言います。この場合、キャンバス上には実際の大きさの要素が、赤で表示されます。



To :制約間の衝突を解決するには

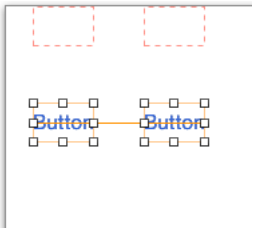
- いずれか一方の制約を削除します。

Tip: 制約を すべて削除して最初からやり直したい場合は、「Issues」＞「Clear Constraints」コマンドを実行してください。

不定性を解決する

制約に**不定性(Ambiguity)**があるとは、それを満たすビューの大きさや配置が何通りもある状況を言います。制約が不足している、ビューの中身の大きさが未定義である、などの原因が考えられます。

この場合Interface Builderは、フレームを橙色の境界線で表示します。詳しくは「Issues」ナビゲータを使ってください。解決方法は原因によって異なります。



制約が不足している場合

単に制約が不足しているという、最も解決が容易な状況です。たとえば、要素の水平位置に関する制約だけで、垂直位置に関する制約がない場合がこれに当たります。解決するためには、欠けている制約を追加する必要があります。

To :欠けている制約を追加するには

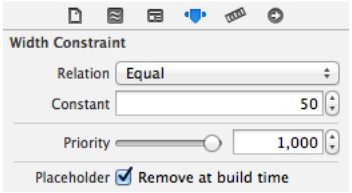
- 「Issues」＞「Add Missing Constraints」コマンドを実行してください。

中身の大きさが未定義である場合

スタックビューなど一部のコンテナビューは、中身の大きさに応じて実行時に自分自身の大きさを決めます。したがって設計時には大きさが決まらないことになります。これは、たとえばビューの最小幅を決める、プレイスホルダ制約を設定することにより対処できます（このプレイスホルダはビルド時、すなわちビューの大きさが決まる時点で削除されます）。

To :プレイスホルダ制約を作成するには

1. 通常通りに制約を追加します。
2. この制約を選択し、「Attributes」インスペクタを開いてください。
3. 「Placeholder」チェックボックスをオンにします。



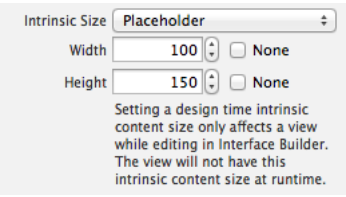
カスタムビューの中身の大きさが分からない場合

標準ビューと違い、カスタムビューには（その中身によって決まる）固有の寸法というものがありません。設計の際、Interface Builderにはこのビューの大きさをどう扱えばよいのか分からないのです。これは、**固有の寸法を決めるプレイスホルダ**を設定することにより、そのビューの中身の大きさが分かるようにして対処できます。

To :固有の寸法を決めるプレイスホルダをビューに設定するには

1. ビューを選択し、「Size」インスペクタを開きます。

2. 「Intrinsic Size」メニューから「Placeholder」を選択してください。
3. 適切な幅と高さの値を設定します。



コード上でデバッグする

大雑把に言うと、レイアウト上の問題のデバッグは2段階に分けて行います。

1. 誤った位置に配置されるビューを認識し、これをもとに不適切な制約を見出す。
2. 不適切な制約をもとに、不適切なコードを見出す。

To :Auto Layoutに関する問題点をデバッグするには

1. 枠が不適切なビューを特定します。

   どのビューに問題があるかは、見れば明らかかもしれません。しかしそうでない場合は、`NSView`の`_subtreeDescription`メソッドでビュー階層記述（テキスト形式）を生成すると役に立つでしょう

   > **Important:** `_subtreeDescription`は、公開APIではありませんが、デバッグに用いることは可能です。

2. 可能であれば、Instrumentsの「Auto Layout」テンプレート上で実行し、問題を再現してください
3. 不適切な制約を特定します。

   あるビューに影響を与える制約は、OS Xならば`constraintsAffectingLayoutForOrientation:`メソッド、iOSならば`constraintsAffectingLayoutForAxis:`メソッドで取得できます。

   制約はデバッガで調べることができます。視覚的書式言語で制約を表示するようになっています（“視覚的書式言語” （42 ページ）を参照）。ビューに識別子があれば、次のように表示されます。

   ```
   <NSLayoutConstraint: 0x400bef8e0
   H:[refreshSegmentedControl]-(8)-[selectViewSegmentedControl] (Names:
   refreshSegmentedControl:0x40048a600,
   selectViewSegmentedControl:0x400487cc0 ) >
   ```

   ない場合は次のようになります。

   ```
   <NSLayoutConstraint: 0x400cbf1c0
   H:[NSSegmentedControl:0x40048a600]-(8)-[NSSegmentedControl:0x400487cc0]>
   ```

4. この時点で、どの制約に問題があるか明らかでない場合は、`visualizeConstraints:`メソッドで制約をウインドウに渡し、画面に表示してください。

制約をクリックすると、その内容がコンソールに表示されます。このようにして、どの制約に問題があり、何が誤っているのかを特定していきます。

この時点で、レイアウトが不定であると表示されるかもしれません。

5. 誤りのあるコードを見つけます。

   不適切な制約を特定できた時点で、修正方法もすぐに分かるかもしれません。

   そうでない場合は、Instrumentsで制約のポインタ（またはその記述）を検索してください。当該制約のライフサイクルにかかわる、生成、修正、ウインドウへの組み込みなどのイベントが表示されます。それぞれについて、実際に発生した箇所のバックトレースも参照できます。想定通りになっていない処理を見つけ、バックトレースを調べてください。ここに問題のコードがあるはずです。

Auto Layoutは、制約を満たせない場合でも大きくは破綻しないように配置する

どうやっても満たせない制約は、プログラムのエラーと考えられます。しかしこのような場合でも、Auto Layoutシステムは、配置が大きく崩れてしまわないよう試みます。

1. `addConstraint:`メソッド（または`NSLayoutConstraint`の`setConstant:`メソッド）は、相互に矛盾する制約があればログに記録し、（プログラムの誤りなので）例外を投げるようになっています。
2. システムは直ちに例外を捕捉します。

   満たすことのできない制約を追加するのはプログラム上のエラーですが、現実には起こりうるものと想定して、クラッシュすることなく適切に対処できるよう、システムが備えておくべきことです。開発者が問題点に気づくよう例外を投げますが、システムがなお機能していれば制約システムのデバッグは容易なので、すぐに例外を捕捉するようになっているのです。
3. システムはレイアウト処理を続行できるよう、相互に矛盾する制約群の中からいずれかを選択し、優先度を「必須」から落とします。落とすと言っても、アプリケーション側で指定できる他の優先度よりは高い、（システム内部のみに存在する）優先度です。その結果、状況が変わり、システムは、（必須でない制約の中では）ここで優先度を落とした制約をまず満たそうと試みるようになります（注意：次の例では、実際に問題が起こっている様子を示すため、制約は当初、すべて必須になっています）。

```
2010-08-30 03:48:18.589 ak_runner[22206:110b] Unable to simultaneously satisfy
constraints:
```

```
(
    "<NSLayoutConstraint: 0x40082d820 H:[NSButton:0x4004ea720'OK']-(20)-|   (Names:
 '|':NSView:0x4004ea9a0 ) >",
    "<NSLayoutConstraint: 0x400821b40 H:[NSButton:0x4004ea720'OK']-(29)-|   (Names:
 '|':NSView:0x4004ea9a0 ) >"
)


Will attempt to recover by breaking constraint
<NSLayoutConstraint: 0x40082d820 H:[NSButton:0x4004ea720'OK']-(20)-|   (Names:
'|':NSView:0x4004ea9a0 ) >
```

Auto Layoutの使用例

Auto Layoutを利用すれば、配置にかかわる複雑な問題の多くを自動的に解決できます。個別にビューを操作する必要はありません。制約をうまく組み合わせれば、従来のようにコードで管理するのは難しい配置も実現できます。向きや大きさの変化に応じてビューが等間隔になるよう並べ直す、スクロールビュー内に、その中身の大きさを調整する要素を置く、あるいはスクロールしても常に同じ位置にとどまる要素を設ける、などが考えられます。

スクロールビューにAuto Layoutを適用する

Auto Layoutを利用してアプリケーションを開発する場合、スクロールビューの取り扱いにはいくつか問題があります。中身の大きさを適切に設定しなければ、スクロールして全体を見ることができません。また、スクロールビューの一番上に目盛や凡例を固定する（スクロールしてもこの要素だけは動かないようにする）のも困難です。

スクロールビューの中身の大きさを制御する

スクロールビューの中身の大きさは、子孫要素の制約によって決まります。

To :スクロールビューの大きさを設定するには

1. スクロールビューを作成します。
2. その内部にUI要素を配置します。
3. 制約を追加して、スクロールビューの中身の幅と高さが完全に決まるようにしてください。

スクロールビュー内のサブビューすべてについて制約が必要です。たとえば、固有の寸法がないビューに制約を定義する場合、前側（leading）に関する制約だけでは不充分です。後側（trailing）、幅、高さの制約も追加してください。スクロールビューの一方の辺からもう一方まで、欠けている制約があってはなりません。

スクロールビュー内にアンカつきのビューを作成する

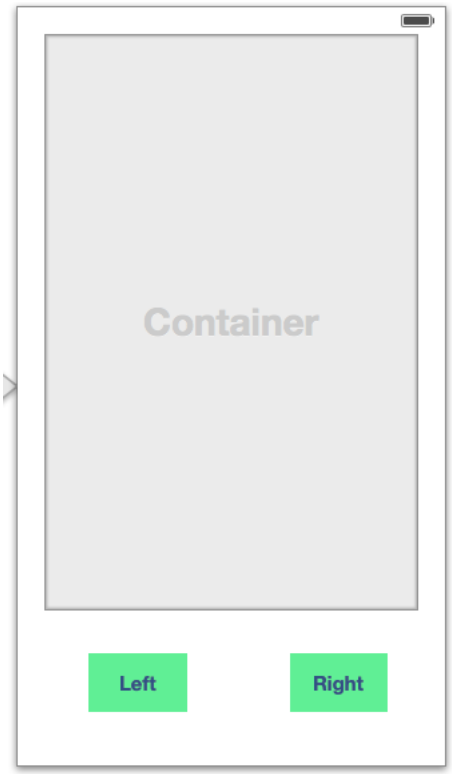
スクロールビュー内に、スクロールしても動かない領域を作りたいことがあります。これは**コンテナビュー**を別に用意して実現できます。

To :スクロールビュー内に別のビューを固定するには

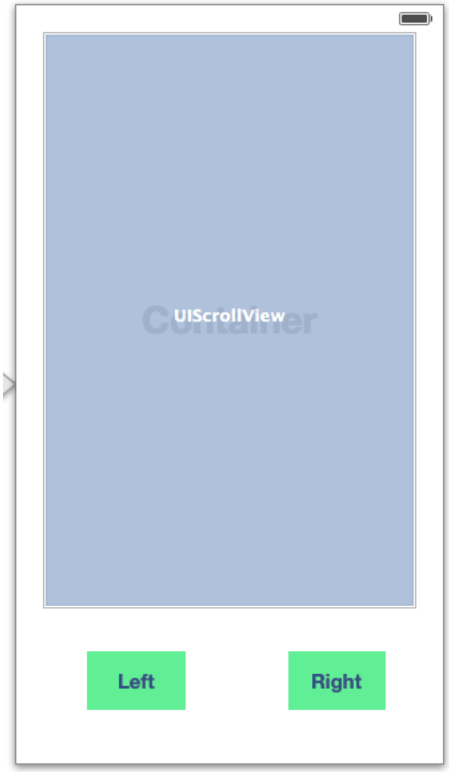
1. スクロールビューを収容するコンテナビューを作っておきます。
2. スクロールビューを作成し、コンテナビュー内に配置します。各辺からの距離はすべて0ポイントとしてください。
3. サブビューを作成し、スクロールビュー内に配置します。
4. サブビューからコンテナビューへの制約を作ります。

上記の手順で、スクロールビュー内にテキストビューを置く例を示します。これはスクロールビューの下辺に固定され、中身をスクロールしても動きません。

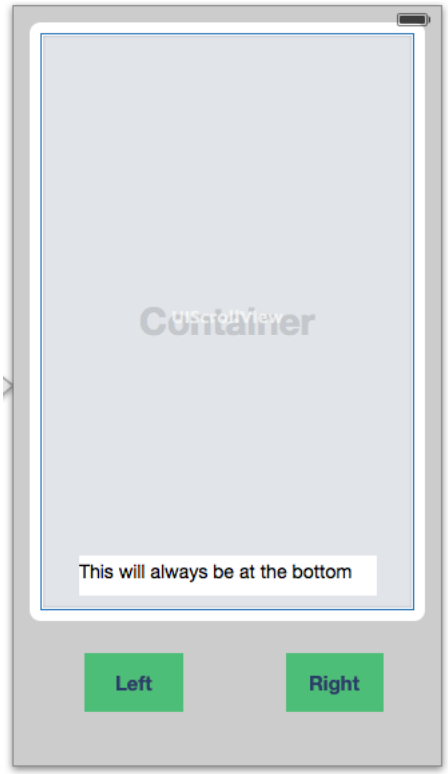
まずコンテナビューを作成します。この中にスクロールビューを収容することになります。その大きさは、スクロールビューに設定する予定の大きさと同じにしてください。



次にスクロールビューを作成し、コンテナビュー内に配置します。辺がすべて、コンテナビューの対応する辺とちょうど重なるよう、距離を0と設定してください。

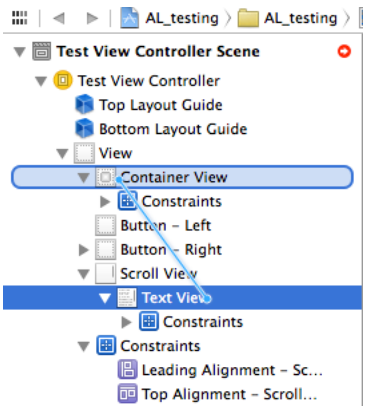


もうひとつビューを作成し、スクロールビュー内に配置します。この例ではテキストビューを置いています。

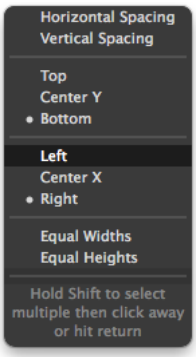


次に、テキストビューからコンテナビューへの制約を作ります。（ビュー階層上、間に入るスクロールビューを飛ばして、）テキストビューをコンテナビューに固定する制約を追加したので、その位置はコンテナビューを基準として決まり、スクロールしても動きません。

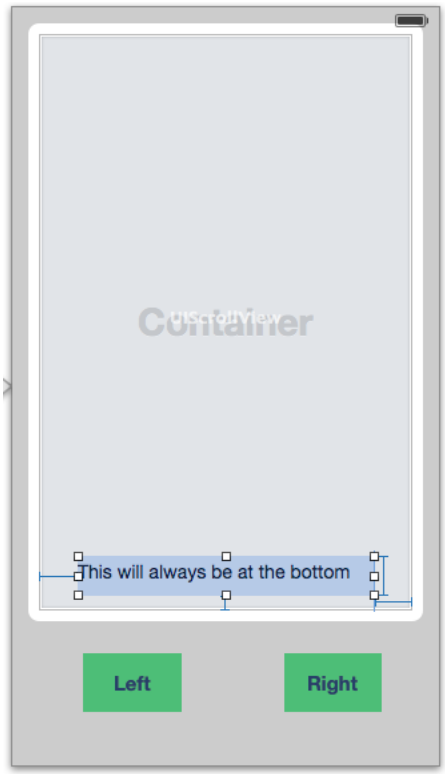
ビュー階層上で複数のビューをまたがる制約を作る場合、Interface Builderのアウトラインビューで、Controlキーを押しながら、該当するビューを摑んでコンテナビューにドラッグする、という方法が通常は簡単でしょう。



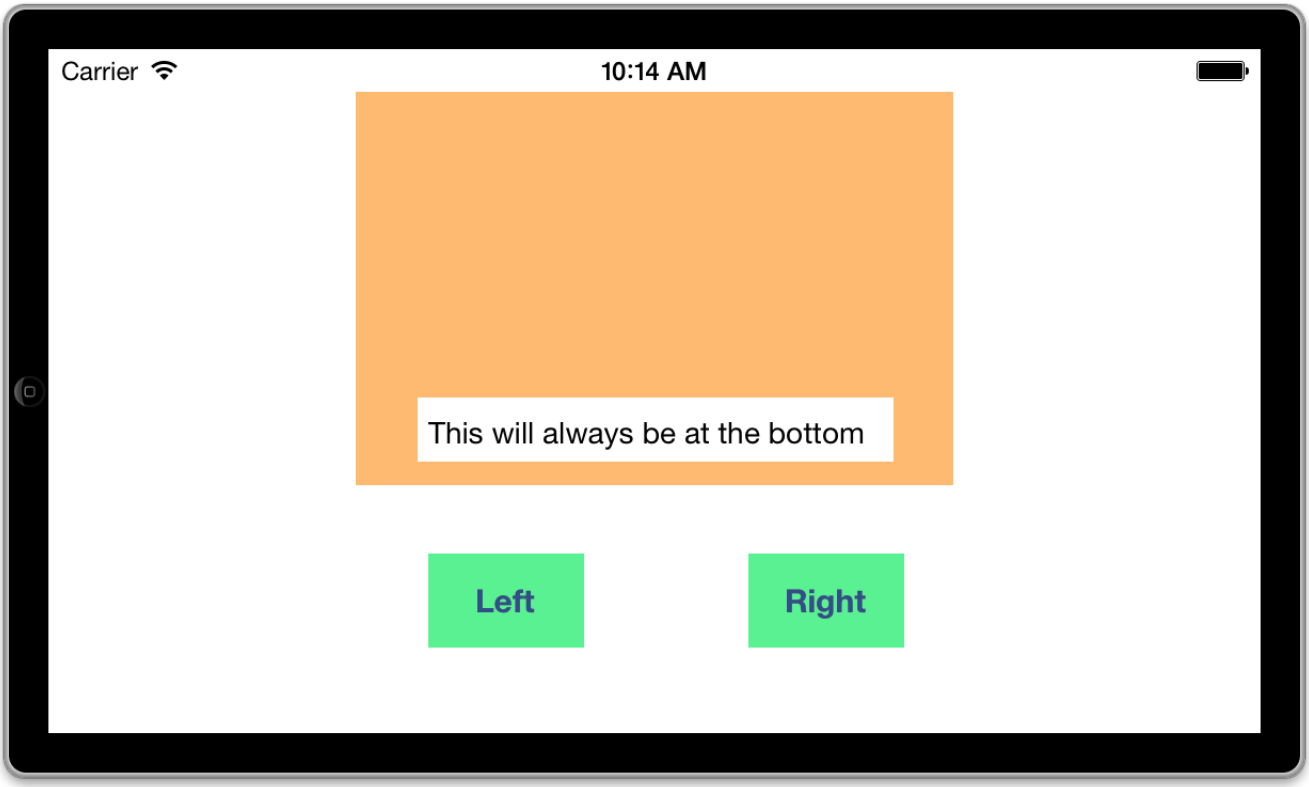
制約の選択肢が現れるので、ビューに応じて適切な制約を設定してください。

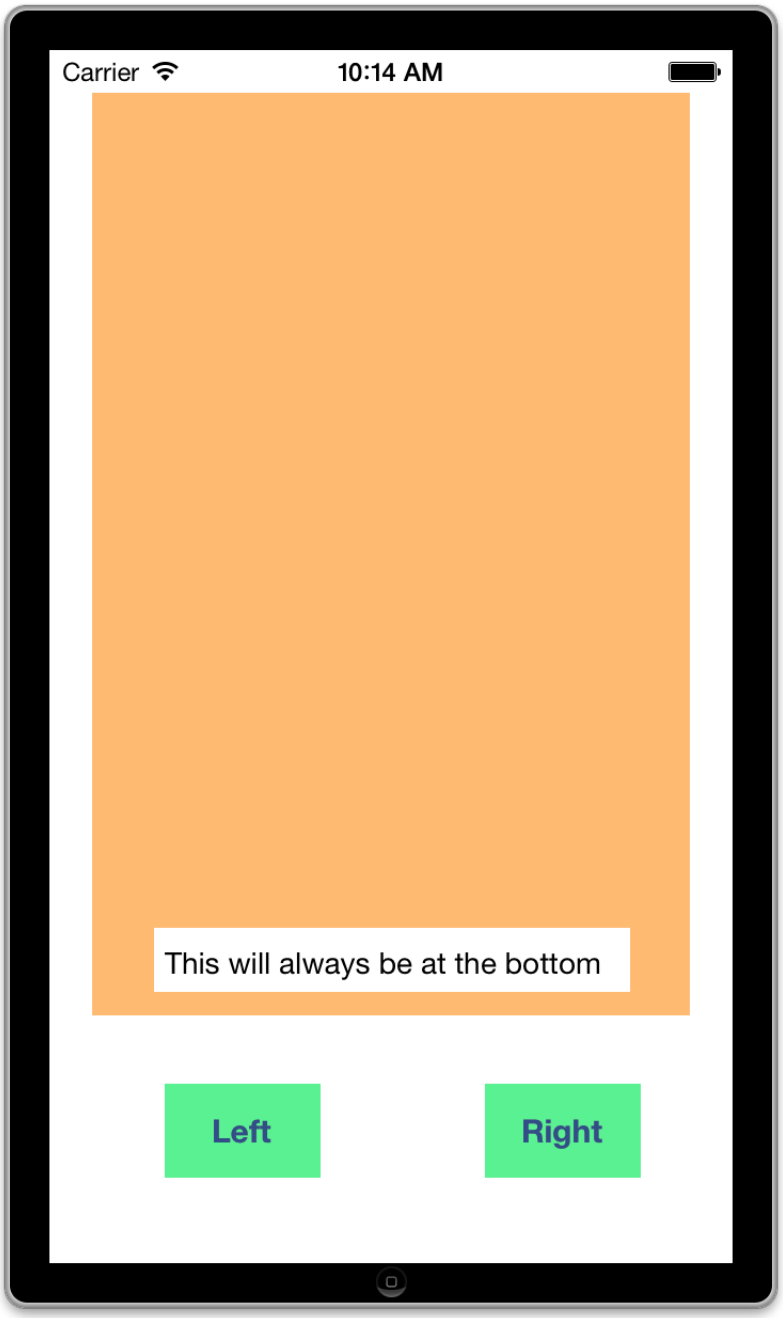


この例では、テキストビューの前側（leading）、後側（trailing）、下側（bottom）から、コンテナビューへの制約を作っています。テキストビューの高さも制約の対象です。



画面が縦の場合と横の場合に、表示がどのようになるかを示します（iOSシミュレータで実行した様子）。テキストビューはスクロールビューの下辺に固定されており、スクロールしても動きません。





間隔の調整

Auto Layoutには、自動的にビューの間隔を調整し、中身に応じて要素の大きさを変える、常套的な設定方法がいくつかあります。以下の各節では、デバイスの向きによらず、ビューを等間隔で並べる制約を作る方法を紹介します。

ビューを等間隔で並べる

デバイスの向きによらず、いくつかのビューを等間隔で並べるためには、間に間隔調整用のスペーサビューを作ります。その制約を適切に設定すれば、向きが変わっても等間隔のままになります。

To :ビューを等間隔で並べるには

1. （実際に画面に現れる）本体ビューをいくつか作ります。
2. それよりもひとつ多い数のスペーサビューを作ります。
3. スペーサビューを端に置き、本体ビューと交互に並べてください。

   画面の左側から右に向かって、次のようなパターンで並ぶことになります。

   `spacer1`|`view1`|`spacer2`|`view2`|`spacer3`。

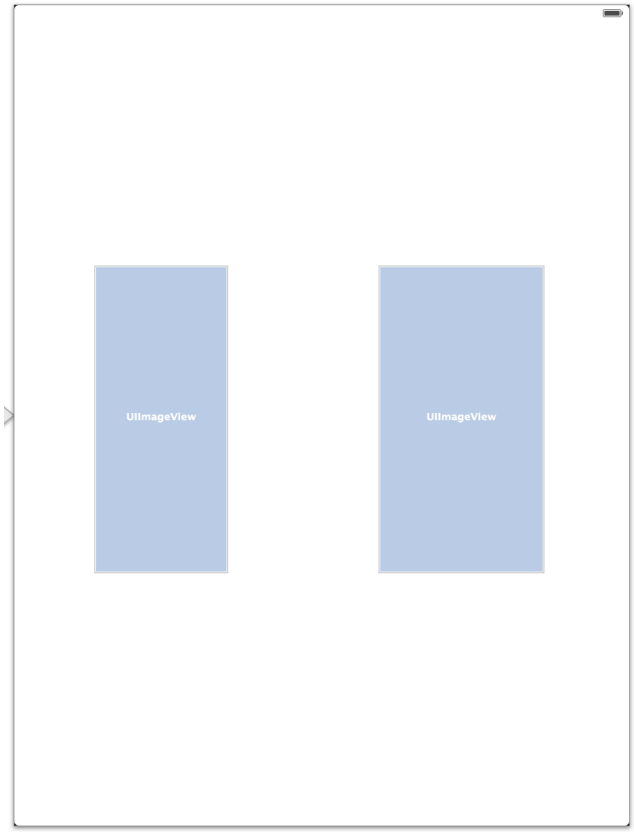
4. スペーサビューに制約を設けて、いずれも長さが同じになるようにします。

   > **注意:** スペーサビューの高さはどうであっても（0でも）構いません。ただし、本体ビューの高さについては制約が必要です。不定のままにしておいてはなりません。

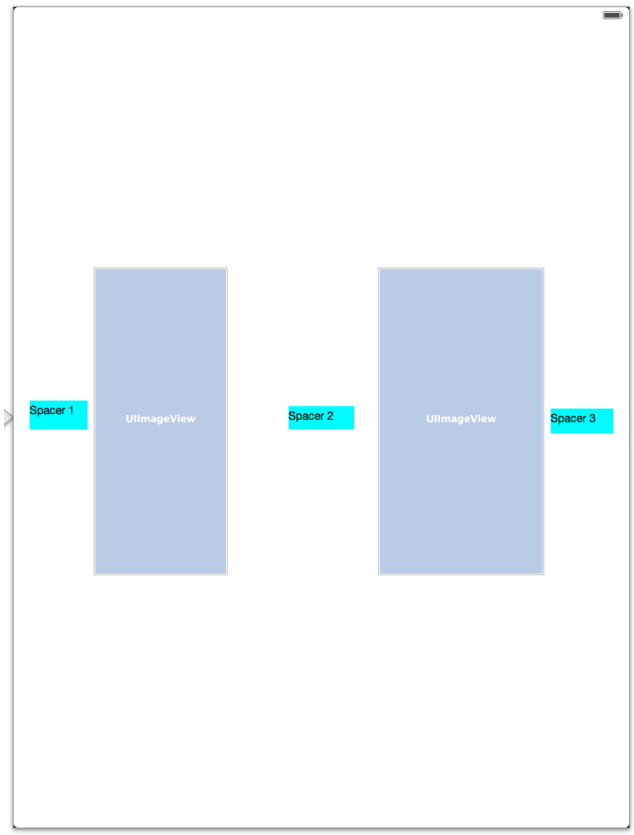
5. 先頭のスペーサビューからコンテナビューへの前側（leading）制約を作ります。
6. 末尾のスペーサビューからコンテナビューへの後側（trailing）制約を作ります。
7. さらに、スペーサビューと本体ビューの間にも制約を作ります。

> **注意:** ビューを縦に並べる場合は、画面の上から、下に向かって順に並べてください。その上で、スペーサビューの高さをすべて同じにします。

以上の手順で、2つのビューを等間隔で並べる例を示します。説明のため、スペーサビューも目に見えるようにしてありますが、通常は背景も中身もなしにしてください。まず、ビューを2つ作ってストーリーボードに配置します。



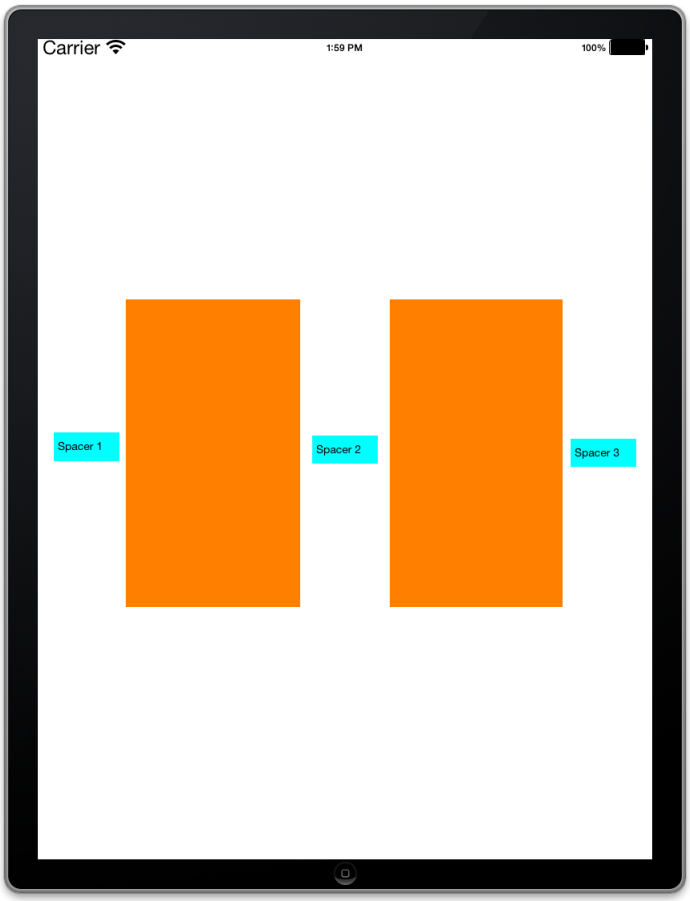
スペーサビューを3つ作り、左側ビューの左、2つのビューの間、右側ビューの右にそれぞれ配置します。この時点では、スペーサビューの大きさが同じでなくても構いません。後で制約を設定すれば大きさが決まります。

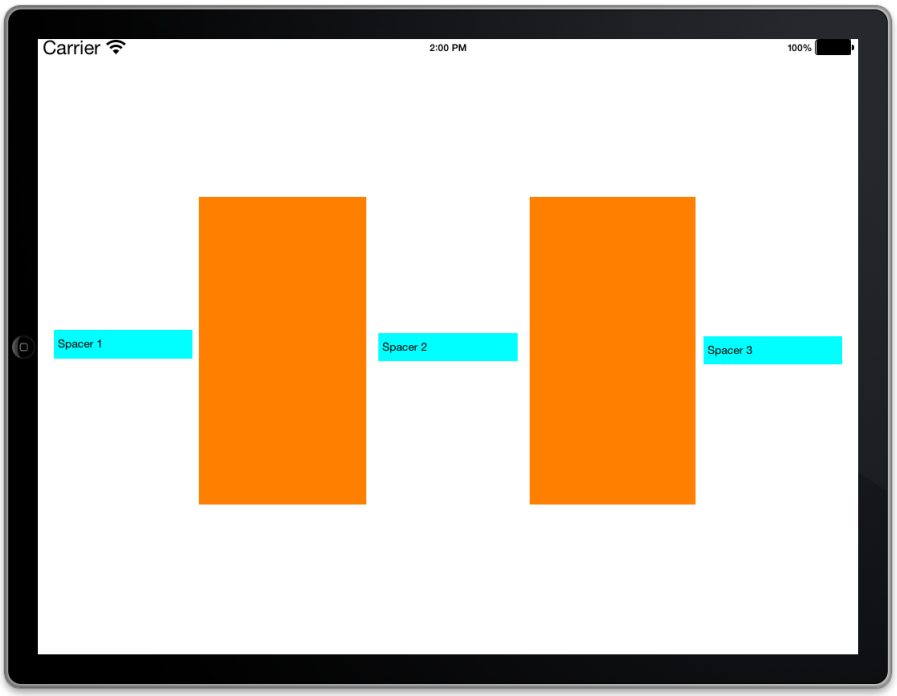


スペーサビューに次の制約を設定してください。

- スペーサビュー2および3の幅を、スペーサビュー1と同じにする制約。
- スペーサビュー1の幅を、ある最小幅以上にする制約。
- スペーサビュー1の前側からコンテナへの制約（Leading Space to Container）。
- スペーサビュー1からビュー1への水平間隔制約（Horizontal Spacing）。これを優先度1000の「less-than-or-equal-to」制約とします。
- スペーサビュー2からビュー1およびビュー2への水平間隔制約（Horizontal Spacing）。これを優先度999の「less-than-or-equal-to」制約とします。
- スペーサビュー3からビュー2への水平間隔制約（Horizontal Spacing）。これを優先度1000の「less-than-or-equal-to」制約とします。
- スペーサビュー3の後側からコンテナへの制約（Trailing Space to Container）。

以上の制約により、可視の本体ビューが2つ、不可視のビュー（スペーサビュー）が3つできます。スペーサビューの大きさはデバイスの向きに応じて自動的に変わり、次の2つの図のように、いつでも本体ビューが等間隔で並びます。





Auto Layoutの働きにより起こる変化をアニメーション表示する

Auto Layoutの働きにより起こる変化のアニメーション表示を完全に制御するためには、制約の変化をプログラムで起こす必要があります。基本的な考え方はiOSでもOS Xでも同じですが、小さな違いが2つあります。

iOSアプリケーションの場合、コードは次のような形になります。

```
[containerView layoutSubtreeIfNeeded]; // 保留になっていたレイアウト処理を確実に実行する
[UIView animateWithDuration:1.0 animations:^{
    // 制約の変化をすべてここで起こす
    [containerView layoutSubtreeIfNeeded]; // 部分木アニメーションブロックのレイアウトを強制し、フレームの変化をすべて取り込む
}];
```

一方、OS Xの場合、レイヤつきアニメーションを使うコードは次のようになります。

```
[containterView layoutIfNeeded];
NSAnimationContext runAnimationGroup:^(NSAnimationContext *context) {
```

```
    [context setAllowsImplicitAnimation: YES];
    // 制約の変化をすべてここで起こす
    [containerView layoutIfNeeded];
}];
```

レイヤつきでなければ、制約のアニメーション表示には、当該制約のアニメータを使わなければなりません。

```
[[constraint animator] setConstant:42];
```

Auto Layoutと連携して動作するカスタムビューの実装

Auto Layoutを活用すれば、ビューが自分自身で配置を管理するようになるので、コントローラクラスに面倒な処理を実装する必要がなくなります。カスタムビュークラスを実装する際には、Auto Layoutシステムに充分な情報を与えて、制約を適切に満たせるようにしなければなりません。特に、ビューに固有の寸法があるか考慮し、ある場合は`intrinsicContentSize`が適切な値を返すように実装する必要があります。

ビューは自分自身の固有の寸法を指定する

ボタンなど、リーフレベルのビューは通常、あるべきサイズに関して、位置決めをするコードよりもよく知っています。この情報をレイアウトシステムに知らせる役割を担うのが`intrinsicContentSize`メソッドです。ビューに固有の寸法がある場合はその具体的な値、ない場合はその旨を返します。

典型的な例として、単一行のテキストフィールドを考えてみましょう。レイアウトシステムは、表示されるテキストの中身には関与しません。知らなければならないのは、当該ビューの中に「何か」があること、そして、その「何か」を（一部を切り抜くことなく）表示するためには、所定の領域を占めることだけです。レイアウトシステムは、`intrinsicContentSize`を呼び出してこの情報を取得し、（1）ビュー中の「何か」を圧縮（縮小や切り抜き）することはできない、（2）ビューは「何か」をできるだけ小さく内包しようとする、という制約を設定します。

ビューには`intrinsicContentSize`メソッドを実装できます。幅と高さを具体的な数値で返すか、幅と高さの一方または両方について、固有の寸法がないことを表す`NSViewNoInstrinsicMetric`を返すことになります。

さらに、次のような`intrinsicContentSize`の実装を考えてみましょう。

- ボタンに固有の寸法は、表示するラベルや画像によって決まります。ボタンの`intrinsicContentSize`メソッドは、ラベルや画像を完全に表示できる幅と高さを返さなければなりません。
- 水平方向のスライダには固有の高さがありますが、固有の幅はありません。スライダの図形は、「最適な幅」というものを想定せずに描画することになります。水平の`NSSlider`オブジェクトは、「`{NSViewNoInstrinsicMetric, <slider height>}`」という値を設定した`CGSize`を返します。スライダを使う側が、幅を決めるための制約を与える必要があります。

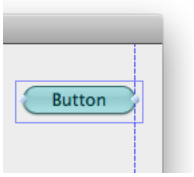
- コンテナ（たとえばNSBoxのインスタンス）には固有の幅や高さがないので、メソッドは{ NSViewNoInstrinsicMetric, NSViewNoInstrinsicMetric }を返すことになります。（コンテナに収容される）サブビューには固有の寸法があるかもしれませんが、ボックス自身にとって固有なのではありません。

固有の寸法が変化したとき、ビューはAuto Layoutシステムにその旨を通知する

さらに、ビューのプロパティ値が変化し、その結果、固有の寸法も変わった場合、invalidateIntrinsicContentSizeを呼び出して、レイアウトシステムにその旨を通知し、配置を計算し直す機会を与えなければなりません。たとえばテキストフィールドは、その文字列の値が変われば`invalidateIntrinsicContentSize`を呼び出さなければなりません。

レイアウトはフレームではなく外接矩形(Alignment Rectangle)を基準とする

レイアウトに関する限り、コントロール部品の枠よりも、目に見えている領域の大きさの方が重要です。したがって、影や溝線のような「飾り」は、レイアウトを決める際には無視するのが普通です。Interface Builderもキャンバス上にビューを配置する際にはこれを無視します。次の例で、Aquaのガイド（青の破線）はボタンの可視領域に沿っています。ボタンの枠（青い実線の矩形）ではありません。



枠ではなく可視領域を基準に配置できるようにするため、ビューは配置の基準となる外接矩形を、レイアウトシステムに渡すようになっています。OS　Xの場合、NSViewShowAlignmentRectsの値をYESとすれば、外接矩形が画面に表示されるので、この矩形が適切かどうか確認できます。

ビューにベースラインオフセットを設定する

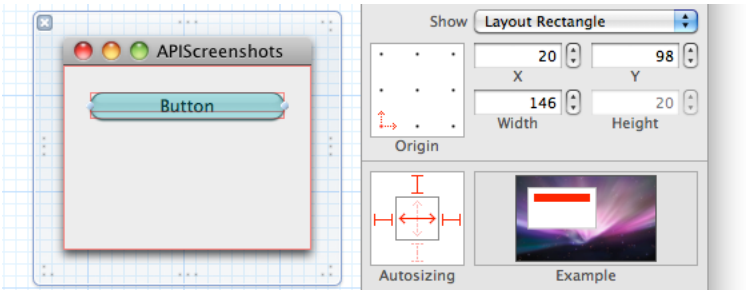
コントロール部品をベースラインに揃えたいことも少なくありません。Interface Builderでは従来から設定できたのですが、プログラムでこの揃え方をすることはできませんでした。現在では、制約として`NSLayoutAttributeBaseline`を指定することにより、実現できるようになっています。

`baselineOffsetFromBottom`メソッドは、`NSLayoutAttributeBottom`と`NSLayoutAttributeBaseline`の距離を返します。`NSView`の場合、この距離のデフォルト値は`0`です。サブクラスでは必要に応じてメソッドをオーバーライドしてください。

Auto Layoutの導入

Auto Layout対応のビューは、そうでないビューが含まれるウインドウと共存できます。したがって、既存のプロジェクトを徐々に、Auto Layoutを採用したものに更新していけます。アプリケーション全体を一度に書き直す必要はありません。プロパティ`translatesAutoresizingMaskIntoConstraints`を使って、ビューをひとつずつ、Auto Layoutを用いる形に書き替えていけばよいのです。

この値が`YES`（デフォルト値）であれば、ビューの自動リサイズマスクを制約の形に置き換えるようになっています。たとえば、あるビューが次のようになっているとき、`translatesAutoresizingMaskIntoConstraints`を`YES`とすれば、「`|-20-[button]-20-|`」および「`V:|-20-[button(20)]`」という制約が、このビューのスーパービューに追加されます。その結果、制約を特に意識していなくても、10.7版より前のOS Xと同じように動作するのです。



ボタンを`15`ポイント左に移動する（実行時に`setFrame:`を呼び出す、という方法で移動しても可）と、「`|-5-[button]-35-|`」および「`V:|-20-[button(20)]`」という制約に変わります。

Auto Layoutに対応済みのビューは、多くの場合、`translatesAutoresizingMaskIntoConstraints`を`NO`とすることになるでしょう。自動リサイズマスクを変換して得られる制約では、スーパービューの枠が与えられれば、当該ビューの枠が完全に決まってしまい、一般に過剰な制約だからです。たとえば、ラベルを変えれば幅が自動的に変わって欲しいと思っても、この制約が邪魔になってしまいます。

`setTranslatesAutoresizingMaskIntoConstraints:`を実行するべきでない状況としては、ビューとそのコンテナとの関係を、開発者自身が指定できない場合が考えられます。たとえば、`NSTableRowView`のインスタンスを、`NSTableView`の脇に配置したいとします。これは、自動リサイズマスクを制約に変換するだけで可能かもしれませんし、そうでないかもしれません。具体的な実装方法によって異なるのです。`setTranslatesAutoresizingMaskIntoConstraints:`を実行するべきでないビューとしてはほかに、`NSTableCellView`オブジェクト、`NSSplitView`のサブビュー、`NSTabViewItem`のビュー、あるいは`NSPopover`、`NSWindow`、`NSBox`のコンテンツビューなどがありま

す。ここで、旧Cocoaレイアウトに慣れている開発者に注記しておきましょう。旧スタイルのビューで`setAutoresizingMask:`を実行していなかった場合、Auto Layoutに移行しても、`setTranslatesAutoresizingMaskIntoConstraints:`は実行しないでください。

ビューが`setFrame:`を呼び出して独自にレイアウトを決めている場合、おそらく同じコードがそのまま通用します。独自に配置するビューについて、引数を`NO`として`setTranslatesAutoresizingMaskIntoConstraints:`を呼び出すことは避けてください。

視覚的書式言語

この付録では、Auto Layoutの視覚的書式言語を使って、標準的な間隔や寸法、垂直レイアウト、優先度の異なる制約など、よく使われる制約を指定する方法を説明します。さらに、言語の完全な構文も示します。

視覚的書式文字列の構文

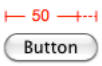
この書式で指定できる制約の例を以下に示します。制約の内容が、見た目で把握できるようになっています。

標準的な間隔

```
[button]-[textField]
```

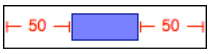
幅の制約

```
[button(>=50)]
```



スーパービューとの接続

```
|-50-[purpleBox]-50-|
```



高さ方向のレイアウト

```
V:[topField]-10-[bottomField]
```

間隔なし（じかに接触）

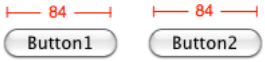
```
[maroonView][blueView]
```

優先度

```
[button(100@20)]
```

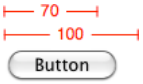
同じ幅

```
[button1(==button2)]
```



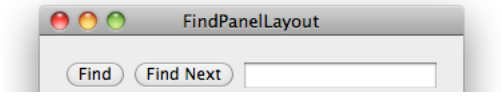
複数の述語

```
[flexibleButton(>=70,<=100)]
```



各要素を一直線に並べたレイアウト

```
|-[find]-[findNext]-[findField(>=20)]-|
```



この記法は、表現力よりも見た目の分かりやすさを重視しています。実際のユーザインターフェイスで有用な制約はほとんど表現できますが、いくつか例外があります。有用だけれどもこの方法で表現できないものとしては、縦横比に関する制約（たとえば「`imageView.width = 2 * imageView.height`」）があります。このような制約を生成するためには、`constraintWithItem:attribute:relatedBy:toItem:attribute:multiplier:constant:`メソッドを使わなければなりません。

視覚的書式文字列の文法

視覚的書式文字列の文法を以下に示します。リテラルは「`code font`」のようなフォントで表します。「**e**」は空文字列です。

シンボル	置換規則
<visualFormatString>	(<orientation>:)? (<superview><connection>)? <view>(<connection><view>)* (<connection><superview>)?
<orientation>	H|V
<superview>	|
<view>	[<viewName>(<predicateListWithParens>)?]
<connection>	**e**|–<predicateList>–|–
<predicateList>	<simplePredicate>|<predicateListWithParens>
<simplePredicate>	<metricName>|<positiveNumber>
<predicateListWithParens>	(<predicate>(,<predicate>)*)
<predicate>	(<relation>)?(<objectOfPredicate>)(@<priority>)?
<relation>	==|<=|>=
<objectOfPredicate>	<constant>|<viewName> *（注意を参照）*
<priority>	<metricName>|<number>
<constant>	<metricName>|<number>
<viewName>	Cの識別子として解析します。 これはビュー辞書のキーで、その値が`NSView`のインスタンスであるものでなければなりません。
<metricName>	Cの識別子として解析します。これはメトリックス辞書のキーで、その値が`NSNumber`のインスタンスであるものでなければなりません。
<number>	「C」ロケールで、`strtod_l`関数を使って数値に変換します。

> **注意:** viewNameがobjectOfPredicateに還元されるのは、述語の対象がビューの幅または高さの場合に限ります。したがって、「[view1(==view2)]」という記述で、view1とview2の幅が同じ、という制約を表せます。

構文上の誤りがあれば、診断メッセージを添えて例外を投げるようになっています。例：

```
Expected ':' after 'V' to specify vertical arrangement
V|[backgroundBox]|
 ^

A predicate on a view's thickness must end with ')' and the view must end with ']'
|[whiteBox1][blackBox4(blackWidth][redBox]|
                                  ^

Unable to find view with name blackBox
|[whiteBox2][blackBox]
                     ^

Unknown relation. Must be ==, >=, or <=
V:|[blackBox4(>30)]|
               ^
```

書類の改訂履歴

この表は「*Auto Layoutガイド*」の改訂履歴です。

日付	メモ
2013-09-18	Xcode 5のAuto Layoutに関する説明を書き直しました。
2012-09-19	iOS Libraryに追加しました。WWDCビデオのリンクを追加しました。
2012-02-16	コードの細かい誤りを修正しました。
2011-07-06	ユーザインターフェイス要素を配置する、制約ベースのシステムについて説明した新規ドキュメント。

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

Apple Japan
〒106-6140 東京都港区六本木 6
丁目10番1号 六本木ヒルズ
http://www.apple.com/jp

Offline copy. Trademarks go here.