第10章 动画

现在我们已经能够编写一组复杂的 React 组件了，接下来我们就来美化一下它们。动画可以让用户体验变得更加流畅与自然，而 React 的 TransitionGroup 插件配合 CSS3 可以让我们在项目中整合动画效果的工作变得易如反掌。

通常情况下，浏览器中的动画都拥有一套极其命令式的 API。你需要选择一个元素并主动移动它或者改变它的样式，以实现动画效果。这种方式与 React 的组件渲染、重渲染方式显得格格不入，因此 React 选择了一种偏声明式的方法来实现动画。

CSS 渐变组（CSS Transition Group）会在合适的渲染及重渲染时间点有策略地添加和移除元素的 class，以此来简化将 CSS 动画应用于渐变的过程。这意味着唯一需要你完成的任务就是给这些 class 写明合适的样式。

间隔渲染以牺牲性能为代价提供了更多的扩展性和可控性。这种方法需要更多次的渲染，但同时也允许你为 CSS 之外的内容（比如滚动条位置及 Canvas 绘图）添加动画。

CSS 渐变组

看一下我们的示例程序——问卷制作工具——是如何在问卷编辑器中渲染问题列表的。

```
<ReactCSSTransitionGroup transitionName='question'>
  {questions}
</ReactCSSTransitionGroup>
```

ReactCSSTransitionGroup 是一款插件，它在文件最顶部通过 `var ReactCSSTransitionGroup = React.addons.ReactCSSTransitionGroup;` 语句被引入。

它会自动在合适的时候处理组件的重渲染，同时根据当前的渐变状态调整渐变组的 class 以便实现组件样式的改变。

> 我们有一个贯穿全书的示例项目，一个问卷制作工具，你可以在 *https://github.com/backstopmedia/bleeding-edge-sample-app* 阅读全部源码。

给渐变 class 添加样式

按照惯例，为元素添加 `transitionName='question'` 意味着给它添加了 4 个 class：question-enter、question-enter-active、question-leave 及 question-leave-active。当子组件进入或退出 ReactCSSTransitionGroup 时，CSSTransitionGroup 插件会自动添加或移除这些 class。

下面是问卷编辑器中使用到的渐变样式：

```
.survey-editor .question-enter {
  transform: scale(1.2);
  transition: transform 0.2s cubic-bezier(.97,.84,.5,1.21);
}

.survey-editor .question-enter-active {
  transform: scale(1);
}

.survey-editor .question-leave {
  transform: translateY(0);
  opacity: 0;
  transition: opacity 1.2s, transform 1s cubic-bezier(.
52,-0.25,.52,.95);
}

.survey-editor .question-leave-active {
  opacity: 0;
  transform: translateY(-100%);
}
```

注意这些 .survey-editor 选择器并不是 ReactCSSTransitionGroup 需要的，它们只是简单

地用来确保这些样式只会在编辑器里生效。

渐变生命周期

question-enter 与 question-enter-active 的区别在于，question-enter 这个 class 是组件被添加到渐变组后即刻添加上的，而 question-enter-active 则是在下一轮渲染时添加的。这样的设计让你能轻松地定义渐变开始时的样式、结束时的样式以及如何进行渐变。

举个例子，当问卷编辑器中的问题被添加到列表时，它们首先被用 scale(1.2) 放大，然后渐变到正常的 scale(1) 状态，总共耗时 0.2 秒。这就创造出了一种你看到的跳出来的效果。

默认情况下，渐变组同时启用了进入和退出的动画，你可以通过给组件添加 transitionEnter={false} 或 transitionLeave={false} 属性来禁用其中一个或全部禁用。除了可以控制选择哪些动画效果外，我们还能根据一个可配置的值在特定的情况下禁用动画，像这样：

```
<ReactCSSTransitionGroup transitionName='question'
  transitionEnter={this.props.enableAnimations}
  transitionLeave={this.props.enableAnimations}>
  {questions}
</ReactCSSTransitionGroup>
```

使用渐变组的隐患

使用渐变组时主要有两个重要的隐患需要注意。

首先，渐变组会延迟子组件的移除直到动画完成。这意味着如果你把一个列表的组件包裹进一个 ReactCSSTransitionGroup 中，却没有为 transitionName 属性指定的 class 明确任何 CSS，这些组件将永远无法被移除——甚至当你尝试不再渲染它们时也不可以。

其次，渐变组的每一个子组件都必须设置一个独一无二的 key 属性。渐变组使用这个属性来判断组件究竟是进入还是退出，因此如果没有设置 key 属性动画可能无法执行，同时组件也会变得无法移除。

注意，即使渐变组只有一个子元素，它也需要设置一个 key 属性。

间隔渲染

使用 CSS3 动画能够获得巨大的性能提升并拥有简洁的代码，但它们并不总是解决问题的正确工具。有些时候你必须要为较老的、不支持 CSS3 的浏览器做兼容，还有些时候你想为

CSS 属性之外的东西添加动画，比如滚动条位置或 Canvas 绘画。在这些情况下，间隔渲染能够满足我们的要求，但是相比 CSS3 动画来说，它会带来一定的性能损耗。

间隔渲染最基本的思想就是周期性地触发组件的状态更新，以明确当前处于整个动画时间中的什么阶段。通过在组件的 render 方法中加入这个状态值，组件能够在每次状态更新触发的重渲染中正确表示当前的动画阶段。

因为这种方法涉及多次重渲染，所以通常最好和 requestAnimationFrame 一起使用以避免不必要的渲染。不过，在 requestAnimationFrame 不被支持或不可用的情况下，降级到不那么智能的 setTimeout 就是唯一的选择了。

使用 requestAnimationFrame 实现间隔渲染

假设你希望使用间隔渲染将一个 div 从屏幕的一边移向另一边，可以通过给它添加 position: absolute 并随着时间变化不停更新 left 或 top 属性来实现。根据消耗时间内的变化总量，用 requestAnimationFrame 来实现这个动画应该可以得出一个流畅的动画。

下面是具体实现的例子。

```
var Positioner = React.createClass({
  getInitialState: function() { return {position: 0}; },

  resolveAnimationFrame: function() {
    var timestamp = new Date();
    var timeRemaining = Math.max(0, this.props.animationComple-
teTimestamp - timestamp);

    if (timeRemaining > 0) {
      this.setState({position: timeRemaining});
    }
  },

  componentWillUpdate: function() {
    if (this.props.animationCompleteTimestamp) {
      requestAnimationFrame(this.resolveAnimationFrame);
    }
  },
```

```
  render: function() {
    var divStyle = {left: this.state.position};
    return <div style={divStyle}>This will animate!</div>
  }
});
```

在这个例子中，组件的 props 中设置了一个名为 animationCompleteTimestamp 的值，它和 requestAnimationFrame 的回调中返回的时间戳一起被用来计算剩余多少位移。计算的结果存在 this.state.position 中，而 render 方法会用它来确定 div 的位置。

由于 requestAnimationFrame 被 componentWillUpdate 方法调用，所以只要组件的 props 有任何的变动（比如改变了 animationCompleteTimestamp）它就会被触发。它又包含了在 resolveAnimationFrame 中的 this.setState 调用。这意味着一旦 animationComleteTimestamp 被设置，组件就会自动调用后续的 requestAnimationFrame 方法，直到当前时间超过了 animationCompleteTimestamp 为止。

注意，这套逻辑只在基于时间戳的情况下成立。对 animationCompleteTimestamp 所做的改变会触发逻辑，而 this.state.position 的值完全依赖于当前时间与 animationCompleteTimestamp 的差。正因如此，render 方法可以自由地在各种动画中使用 this.state.position，包括设置滚动条位置、在 canvas 上绘画，以及任何中间状态。

使用 setTimeout 实现间隔渲染

尽管 requestAnimationFrame 总体上能够以最小的性能损耗实现最流畅的动画，但它在较老的浏览器上是无法使用的，而且它被调用的次数可能比你想象的更频繁（也更加无法预测）。在这些情况下你可以使用 setTimeout。

```
var Positioner = React.createClass({
  getInitialState: function() { return {position: 0}; },

  resolveSetTimeout: function() {
    var timestamp = new Date();
    var timeRemaining = Math.max(0, this.props.animationCompleteTimestamp
     - timestamp);
    if (timeRemaining > 0) {
      this.setState({position: timeRemaining});
    }
  },
```

```
  componentWillUpdate: function() {
    if (this.props.animationCompleteTimestamp) {
      setTimeout(this.resolveSetTimeout, this.props.timeoutMs);
    }
  },

  render: function() {
    var divStyle = {left: this.state.position};

    return <div style={divStyle}>This will animate!</div>
  }
});
```

由于 setTimeout 接受一个显式的时间间隔，而 requestAnimationFrame 是自己来决定这个时间间隔的，因此这个组件需要额外依赖一个变量 this.props.timeoutMs，以此来明确要使用的间隔。

开源库 ReactTweenState 基于这种动画方式提供了一套方便的抽象接口。

总结

使用这些动画技术，你现在可以：

1. 在状态改变过程中，使用 CSS3 和渐变组高效地应用渐变动画。
2. 使用 requestAnimationFrame 为 CSS 之外的东西添加动画，如滚动条位置或 Canvas 绘画。
3. 当 requestAnimationFrame 不被支持时降级到 setTimeout 方法。

在下一章中，你会学到如何优化 React 性能！