新版本 Form 对使用方式进行了简化，因而如果你是从 v3 迁移上来。你可以参考本文做迁移工作。

去除 Form.create

v4 的 Form 不再需要通过 `Form.create()` 创建上下文。Form 组件现在自带数据域，因而 `getFieldDecorator` 也不再需要，直接写入 Form.Item 即可：

```
// antd v3
const Demo = ({ form: { getFieldDecorator } }) => (
  <Form>
    <Form.Item>
      {getFieldDecorator('username', {
        rules: [{ required: true }],
      })(<Input />)}
    </Form.Item>
  </Form>
);

const WrappedDemo = Form.create()(Demo);
```

改成：

```
// antd v4
const Demo = () => (
  <Form>
    <Form.Item name="username" rules={[{ required: true }]}>
      <Input />
    </Form.Item>
  </Form>
);
```

由于移除了 `Form.create()`，原本的 `onFieldsChange` 等方法移入 Form 中，通过 `fields` 对 Form 进行控制。参考示例。

表单控制调整

Form 自带表单控制实体，如需要调用 form 方法，可以通过 `Form.useForm()` 创建 Form 实体进行操作：

```
// antd v3
const Demo = ({ form: { setFieldsValue } }) => {
  React.useEffect(() => {
    setFieldsValue({
      username: 'Bamboo',
    });
  }, []);

  return (
    <Form>
      <Form.Item>
        {getFieldDecorator('username', {
```

```
          rules: [{ required: true }],
        })(<Input />)}
      </Form.Item>
    </Form>
  );
};

const WrappedDemo = Form.create()(Demo);
```

改成：

```
// antd v4
const Demo = () => {
  const [form] = Form.useForm();

  React.useEffect(() => {
    form.setFieldsValue({
      username: 'Bamboo',
    });
  }, []);

  return (
    <Form form={form}>
      <Form.Item name="username" rules={[{ required: true }]}>
        <Input />
      </Form.Item>
    </Form>
  );
};
```

对于 class component，也可以通过 `ref` 获得实体：

```
// antd v4
class Demo extends React.Component {
  formRef = React.createRef();

  componentDidMount() {
    this.formRef.current.setFieldsValue({
      username: 'Bamboo',
    });
  }

  render() {
    return (
      <Form ref={this.formRef}>
        <Form.Item name="username" rules={[{ required: true }]}>
          <Input />
        </Form.Item>
      </Form>
    );
```

```
  }
}
```

由于 Form.Item 内置字段绑定，如果需要不带样式的表单绑定，可以使用 `noStyle` 属性移除额外样式：

```
// antd v3
const Demo = ({ form: { getFieldDecorator } }) => {
  return <Form>{getFieldDecorator('username')(<Input />)}</Form>;
};

const WrappedDemo = Form.create()(Demo);
```

改成：

```
// antd v4
const Demo = () => {
  return (
    <Form>
      <Form.Item name="username" noStyle>
        <Input />
      </Form.Item>
    </Form>
  );
};
```

字段联动调整

新版 Form 采用增量更新方式，仅会更新需要更新的字段。因而如果有字段关联更新，或者跟随整个表单更新而更新。可以使用 `dependencies` 或 `shouldUpdate` 。

onFinish 替代 onSubmit

对于表单校验，过去版本需要通过监听 `onSubmit` 事件手工触发 `validateFields` 。新版直接使用 `onFinish` 事件，该事件仅当校验通过后才会执行：

```
// antd v3
const Demo = ({ form: { getFieldDecorator, validateFields } }) => {
  const onSubmit = e => {
    e.preventDefault();
    validateFields((err, values) => {
      if (!err) {
        console.log('Received values of form: ', values);
      }
    });
  };

  return (
    <Form onSubmit={onSubmit}>
      <Form.Item>
```

```
        {getFieldDecorator('username', {
          rules: [{ required: true }],
        })(<Input />)}
      </Form.Item>
    </Form>
  );
};

const WrappedDemo = Form.create()(Demo);
```

改成：

```
// antd v4
const Demo = () => {
  const onFinish = values => {
    console.log('Received values of form: ', values);
  };

  return (
    <Form onFinish={onFinish}>
      <Form.Item name="username" rules={[{ required: true }]}>
        <Input />
      </Form.Item>
    </Form>
  );
};
```

scrollToField 替代 validateFieldsAndScroll

新版推荐使用 `onFinish` 进行校验后提交操作，因而 `validateFieldsAndScroll` 拆成更独立的 `scrollToField` 方法：

```
// antd v3
onSubmit = () => {
  form.validateFieldsAndScroll((error, values) => {
    // Your logic
  });
};
```

改成：

```
// antd v4
onFinishFailed = ({ errorFields }) => {
  form.scrollToField(errorFields[0].name);
};
```

初始化调整

此外，我们将 `initialValue` 从字段中移到 Form 中。以避免同名字段设置 `initialValue` 的冲突问题：

```
// antd v3
const Demo = ({ form: { getFieldDecorator } }) => (
  <Form>
    <Form.Item>
      {getFieldDecorator('username', {
        rules: [{ required: true }],
        initialValue: 'Bamboo',
      })(<Input />)}
    </Form.Item>
  </Form>
);

const WrappedDemo = Form.create()(Demo);
```

改成：

```
// antd v4
const Demo = () => (
  <Form initialValues={{ username: 'Bamboo' }}>
    <Form.Item name="username" rules={[{ required: true }]}>
      <Input />
    </Form.Item>
  </Form>
);
```

在 v3 版本中，修改未操作的字段 `initialValue` 会同步更新字段值，这是一个 BUG。但是由于被长期作为一个 feature 使用，因而我们一直没有修复。在 v4 中，该 BUG 已被修复。 `initialValue` 只有在初始化以及重置表单时生效。

嵌套字段路径使用数组

过去版本我们通过 `.` 代表嵌套路径（诸如 `user.name` 来代表 `{ user: { name: '' } }`）。然而在一些后台系统中，变量名中也会带上 `.`。这造成用户需要额外的代码进行转化，因而新版中，嵌套路径通过数组来表示以避免错误的处理行为（如 `['user', 'name']`）。

也因此，诸如 `getFieldsError` 等方法返回的路径总是数组形式以便于用户处理：

```
form.getFieldsError();

/*
[
  { name: ['user', 'name'], errors: [] },
  { name: ['user', 'age'], errors: ['Some error message'] },
]
*/
```

嵌套字段定义由：

```
// antd v3
<Form.Item label="Firstname">{getFieldDecorator('user.0.firstname', {})(<Input />)}
</Form.Item>
```

改至：

```
// antd v4
<Form.Item name={['user', 0, 'firstname']} label="Firstname">
  <Input />
</Form.Item>
```

相似的，`setFieldsValue` 由：

```
// antd v3
this.formRef.current.setFieldsValue({
  'user.0.firstname': 'John',
});
```

改至：

```
// antd v4
this.formRef.current.setFieldsValue({
  user: [
    {
      firstname: 'John',
    },
  ],
});
```

validateFields 不再支持 callback

`validateFields` 会返回 Promise 对象，因而你可以通过 `async/await` 或者 `then/catch` 来执行对应的错误处理。不再需要判断 `errors` 是否为空：

```
// antd v3
validateFields((err, value) => {
  if (!err) {
    // Do something with value
  }
});
```

改成：

```
// antd v4
validateFields().then(values => {
  // Do something with value
});
```