Setup Miniprogram

- setData
- IntersectionObserver
- Publish-subscribe
- Subpackages-async
- Enhancer
- etc…

Lei Zhang

Dec. 08, 2023

`setData`

`setData` Tips

👉 更新流程

```
async loadData() {
  const { data } = await request()

  this.setData({ list: data })
}
```

- 逻辑层虚拟 DOM 树的遍历和更新，触发组件生命周期等；
- 将 data 从逻辑层传输到视图层；
- 视图层虚拟 DOM 树的更新、真实 DOM 元素的更新并触发页面渲染更新。

👉 数据通信

```
async loadData() {
  const { list } = this.data

  const { data } = await request()

  this.setData({
    list: [...list, ...data]
  })
}
```

对于第 2 步，数据传输的耗时与**数据量的大小**正相关。

每一次调用 `loadData` 后，`list` 会逐渐变大，

在上拉加载下一页等常规场景中，加载的页数越多，导致渲染的越卡越慢。

👉 优化

```
async loadData() {
  const { list } = this.data

  const { data } = await request()

  this.setData({
    [`list[${list.length}]`]: data
  })
}
```

每次调用 `setData`，数据通信大小都相同，无论分页多少次，都能保证每次高效的渲染。

`setData` Pattern

✅ Best Practices

- `setData` 应只传入发生变化的字段；
- 以**数据路径**形式改变数组中的某一项或对象的某个属性；
- 仅在需要进行页面内容更新时调用 `setData`；
- 对连续的 setData 调用尽可能的进行**合并**；

❌ Incorrect Practices

- 不要在 `setData` 中一次性传所有 `data`；
- 避免全量覆盖数组或对象；
- 避免不必要的 `setData`；
- 避免以过高的频率持续调用 `setData`；

IntersectionObserver

IntersectionObserver Core

IntersectionObserver 对象，用于推断某些节点是否可以被用户看见、有多大比例可以被用户看见。

```
async loadData() {
  const { list } = this.data

  const { data } = await request()

  this.setData({
    [`list[${list.length}]`]: data
  })
}
```

```
<view
  wx:for="{{list}}"
  wx:key="index"
  wx:for-item="sourceData"
>
  <item wx:for="{{sourceData}}" wx:key="*this">
    {{ item }}
  </item>
</view>
```

我们通过以**数据路径**形式将每次分页的数据存储，确保了 `setData` 高效的更新，

但是**视图层**的 `WXML` 节点树仍然会不断的增加，

一个太大的 `WXML` 节点树会增加内存的使用，样式重排时间也会更长，影响体验。

IntersectionObserver

Tips

- `<item />` 的节点树巨大；
- `<item />` 中存在定时器；
- `<item />` 挂载时需要发送请求；
- `<item />` 会在 `onShow` 处理异步实务；

如果它是一个秒杀类型的营促销组件，20 多个秒杀倒计时便会让页面崩溃，

客户端（手机），大部分最多 5 条数据便会撑满整个**视口**，我们可以通过 IntersectionObserver 来判断如果组件在 **视口** 就挂载，反之则卸载，

确保每次挂载的组件只有 5 个，而不是全量显示。

```
const showNum = 0.5; // 上下半屏

const { windowHeight } = wx.getSystemInfoSync();

this.observer = this.createIntersectionObserver();

this.observer
  .relativeToViewport({
    top: showNum * windowHeight,
    bottom: showNum * windowHeight,
  })
  .observe(`#${id}`, (res) => {
    let { intersectionRatio } = res;
    if (intersectionRatio === 0) {
      // 超过预定范围，从页面卸载
      this.setData({ showSlot: false });
    } else {
      // 达到预定范围，挂载进页面
      this.setData({ showSlot: true });
    }
  });
```

Publish–subscribe

Publish-subscribe `Pattern`

Publish-subscribe 模式，用于在小程序种跨页面、组件的通信。

```
App({
  addListener(callback) {
    this.callback = callback;
  },

  setChangedData(payload) {
    if (this.callback) {
      this.callback(payload);
    }
  },
});
```

```
const app = getApp();

Page({
  onLoad() {
    app.addListner((payload) ⇒ {
      this.setData({
        data: payload,
      });
    });
  },
});
```

```
const app = getApp();

Page({
  onClick() {
    app.setChangedData("payload");
  },
});
```

以上是一个跨页通信的 `demo`，只能支持一种 `Event` 的通知，而且也不能针对这个 `Event` 添加多个监听者，一个基本的 Publish-subscribe 需要具备：

- 支持多种 `Event` 的通知；
- 支持对某一 `Event` 可以添加多个监听者；
- 支持对某一 `Event` 可以移除某一监听者；

Publish-subscribe `Core`

一级页面订阅 `on`，二三级页面发布 `emit`，订阅者取消订阅 `off`。

```
const events = {};
```

```
function on(name, self, callback) {
  const tuple = [self, callback];

  const tuples = events[name];

  if (Array.isArray(tuples)) {
    tuples.push(tuple);
  } else {
    events[name] = [tuple];
  }
}
```

```
function emit(name, payload) {
  const tuples = events[name];

  if (Array.isArray(tuples)) {
    tuples.map((tuple) => {
      const [self, callback] = tuple;

      tuples.call(self, payload);
    });
  }
}
```

```
function off(name, self) {
  const tuples = events[name];

  if (Array.isArray(tuples)) {
    events[name] = tuples.filter((tuple) => {
      return tuple[0] !== self;
    });
  }
}
```

Subpackages-async

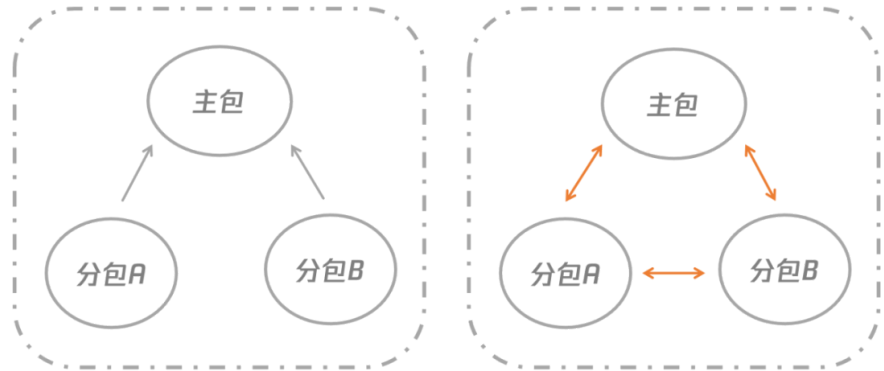
Subpackages-async

Tips

- 第三方库全局使用只能放到主包；
- 公共组件过多，主包代码体积过大；
- 多业务的分包难以划分；
- …

让分包种的组件、模块，能相互的异步引用，

更好对多业务的分包进行划分。



分包异步化之前　　分包异步化之后

Subpackages-async

Core

通过异步分包，异步加载组件、模块。

```
// app.json

{
    root: 'asyncPkgs',
    pages: []
},
```

```
// page.json or component.json

{
  "usingComponents": {
    "foo": "/asyncPkgs/foo/foo"
  },
  "componentPlaceholder": {
    "foo": "view"
  }
}
```

```
<!-- page.wxml or component.wxml -->

<foo />
```

```
// page.js or component.js

// 使用回调函数风格的调用
require(
  "/asyncPkgs/utils.js",
  (utils) => {
    // Wechat MiniProgram
    console.log(utils.whoami);
  });

// 或者使用 Promise 风格的调用
require.async("/asyncPkgs/index.js")
  .then((pkg) => {
    // 'common'
    pkg.getPackageName();
  });
```

Enhancer

Enhancer `Tips`

某些场景，我们可能需要去增强 `Page`，例如：

- 在所有 `Page` 重写生命周期钩子；
- 在所有 `Page` 中填充 `data` 属性；
- 在某些 `Page` 中增加登录校验；
- ...

还记得 `React` 中的 `HOC` 吗?

```
const EnhancedComponent = higherOrderComponent(WrappedComponent)
```

参考 `HOC`，我们可以编写一个 `Enhancer` 来增强小程序的 `Page`：

Enhancer Core

```
const UserEntity from '/entities/user'

const Enhancer = (props, options = {}) ⇒ {
  // overwrite onLoad
  const { onLoad } = props
  onLoad && delete props.onLoad
  props.onLoad = function() {
    // do something what you want ...
    // exec original onLoad
    onLoad && onLoad.apply(this, arguments);
  };

  // check login
  const { checkLogin } = options
  if (checkLogin && !UserEntity.isLogin) {
    wx.redirectTo({ url: '/pages/login' })
  }

  return props
}
```

```
import Enhancer from "/enhancer";

// enhancer page props
const props = Enhancer(
  {
    data: {},

    onLoad(options) {},

    onUnload() {},

    // etc ...
  },
  {
    checkLogin: true,
  }
);

// create page
Page(props);
```

🫶 Thanks!

幻灯片可在 PassionZale/talks 查看与下载