微信商学院　微信支付

小程序优化技巧

目录：

1. 提高页面加载速度
2. 预加载下一访问页面的数据
3. 不依赖任何工具的组件化

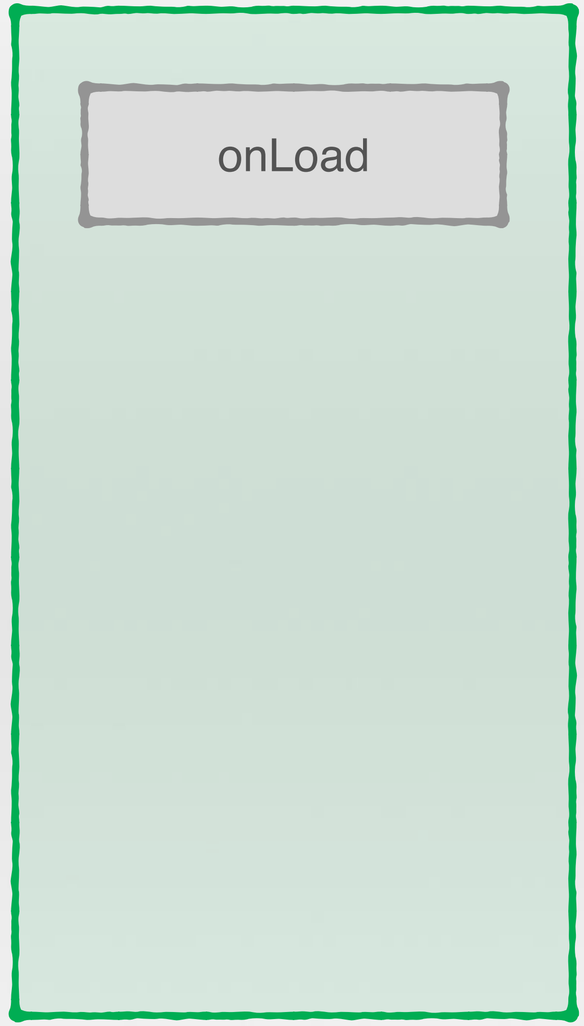
提高页面加载速度

< 前端永恒不变的话题 />

一次常规的页面跳转流程

首页

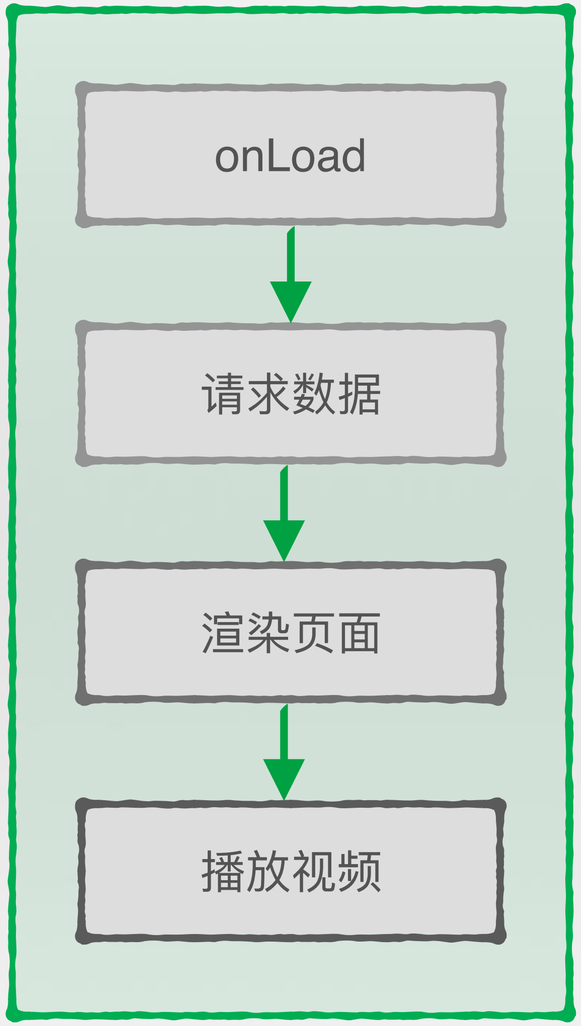
Page(page/play/index)



一次常规的页面跳转流程

首页

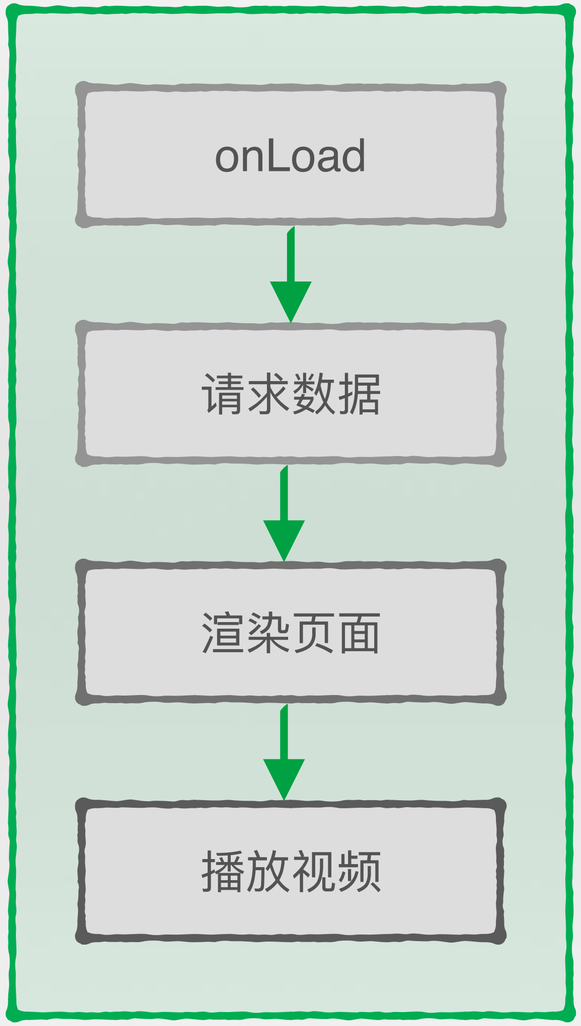
Page(page/play/index)



一次常规的页面跳转流程

首页

Page(page/play/index)

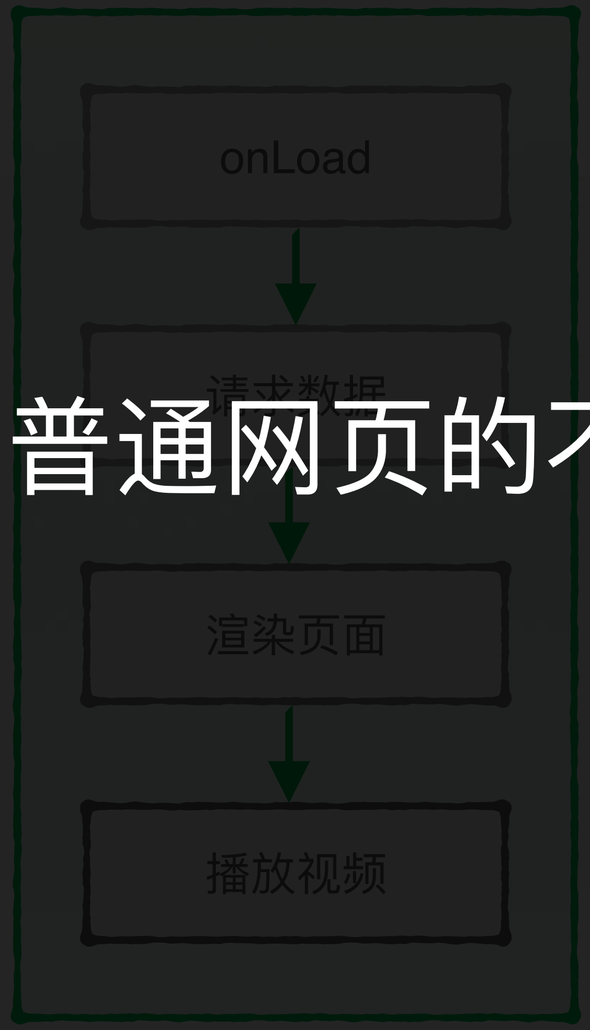


播放页

一次常规的页面跳转流程

首页

Page(page/play/index)



播放页

小程序与普通网页的不同之处?

一次常规的页面跳转流程

首页

播放页

100 ~ 200 ms

onTap( )

onLoad( )

一次常规的页面跳转流程

首页

100 ~ 200 ms

onTap( ) ⟹ onLoad( )

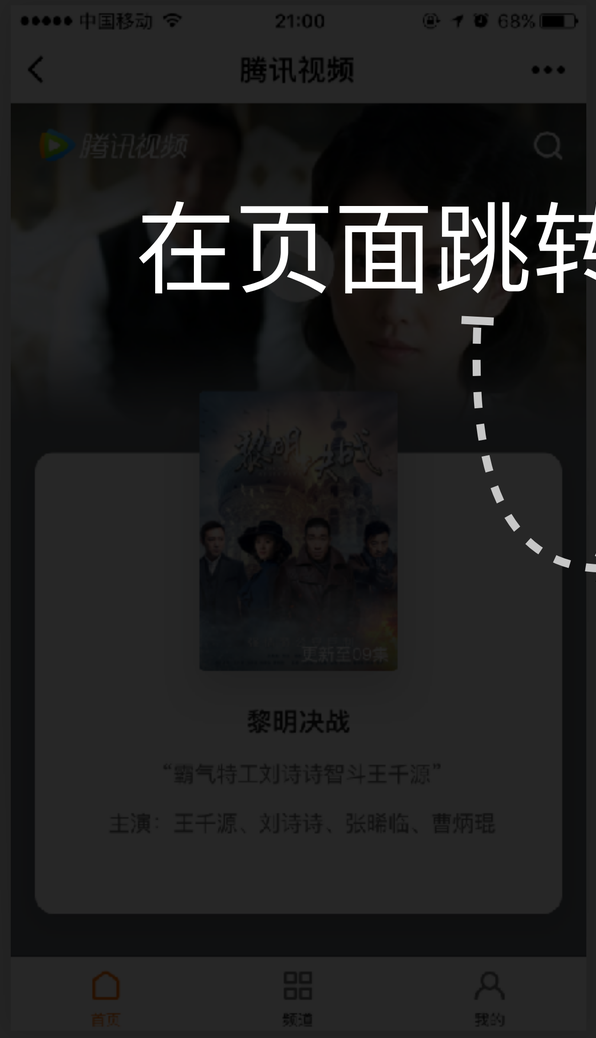
跳过程中，JS仍可执行

播放页

一次常规的页面跳转流程

首页

播放页



在页面跳转时，提前发起下个页面数据请求

100 ~ 200 ms

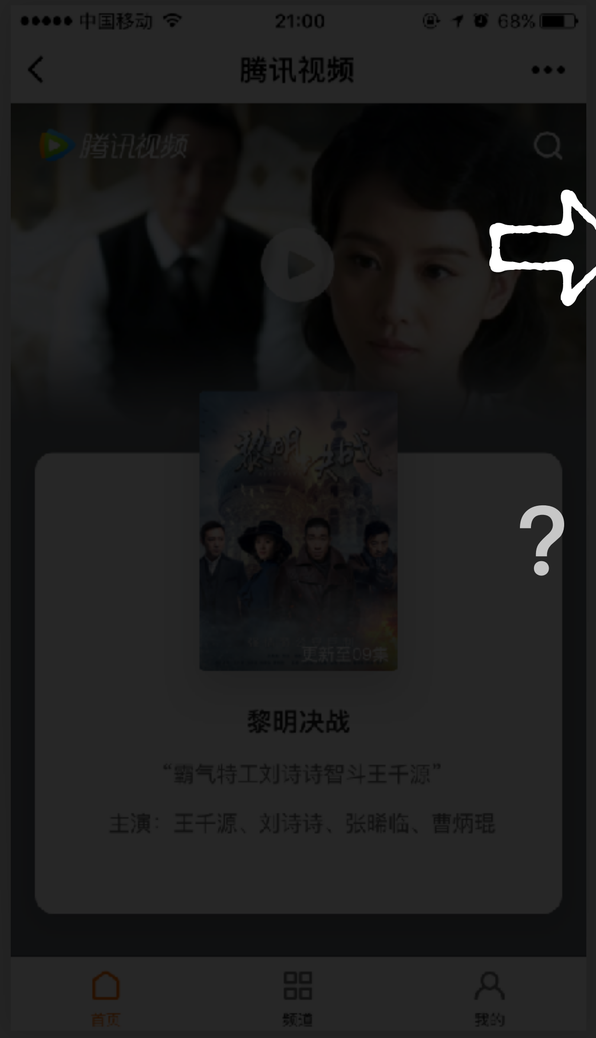
onTap( ) ⟹ onLoad( )

：小程序的 JS 是不间断运行的，不受页面跳转的影响

跳过程中，JS仍可执行

一次常规的页面跳转流程

首页



播放页

⇨ 在 A 页面预加载 B 页面的数据

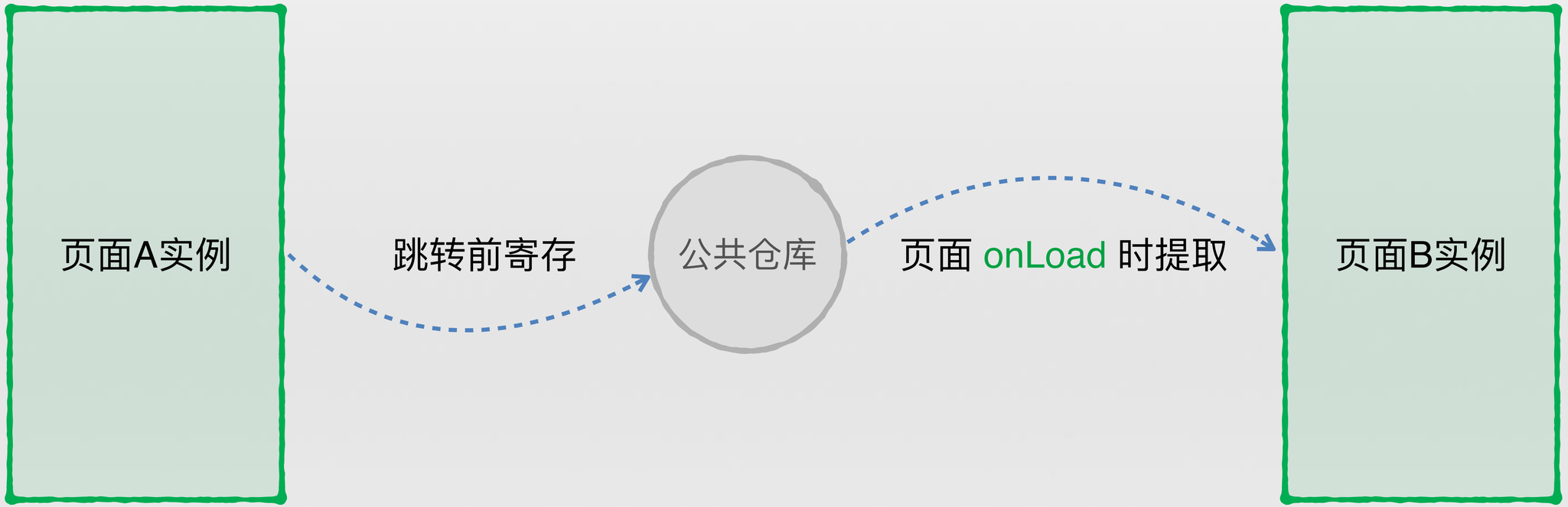
100 ~ 200 ms

onTap( ) ⟹ onLoad( )

？ 如何解决页面间耦合，减低开发的复杂度

⟶ 跳过程中，JS仍可执行

通过中介模块存放临时数据



通过中介模块存放临时数据

封装小程序的跳转方法，以便挂载钩子逻辑

0 ms 用户点击A页面链接：

```
onTap: function (e) {
  this.$route('play?cid='+e.currentTarget.dataset.cid)
},
```

在跳转方法里触发
目标页面 **onNavigate** 方法

0 ms 调用B页面的 onNavigate( ) 方法：

```
onNavigate: function () {
  this.$put('play-data', this.fetchData)
},
fetchData: function () {
  return new Promise(function (resolve) { wx.request({/*...*/}) })
},
```

将预请求以 promise 对象形式寄存

200 ms B页面 onLoad( )：

```
onLoad: function() {
  var p = this.$take('play-data') || this.fetchData()
  p.then(function (data) {
    ...
  })
}
```

页面实例化耗时

0 ms 用户点击A页面链接：

```
onTap: function (e) {
  this.$route('play?cid='+e.currentTarget.dataset.cid)
},
```

0 ms 调用B页面的 onNavigate( ) 方法：

```
onNavigate: function () {
  this.$put('play-data', this.fetchData)
},
fetchData: function () {
  return new Promise(function (resolve) { wx.request({/*...*/}) })
},
```

200 ms B页面 onLoad( )：

```
onLoad: function() {
  var p = this.$take('play-data') || this.fetchData()
  p.then(function (data) {
    ...
  })
}
```

页面实例化耗时

0 ms 用户点击A页面链接：

```
onTap: function (e) {
  this.$route('play?cid='+e.currentTarget.dataset.cid)
},
```

0 ms 调用B页面的 onNavigate( ) 方法：

```
onNavigate: function () {
  this.$put('play-data', this.fetchData)
  return new Promise(function (resolve) { wx.request({/*...*/}) })
},
```

如何扩展页面的生命周期?

200 ms B页面 onLoad( )：

```
onLoad: function() {
  var p = this.$take('play-data') || this.fetchData()
  p.then(function (data) {
    ...
  })
}
```

小程序如何创建页面实例

程序启动：

调用 Page(obj) 函数注册页面

代码来自开发者工具

```
pageHolder = function(e) {
  if (!__wxRouteBegin)
      throw (0,
      c.error)("Page 注册错误", "Pl
      new i.AppServiceEngineKnown
  __wxRouteBegin = !1;
  var t = __wxConfig.pages
    , n = t[g];
  if (g++,
  "Object" !== (0,
  c.getDataType)(e))
      throw (0,
      c.error)("Page 注册错误", "Op
      new i.AppServiceEngineKnown
  (0,
  c.info)("Register Page: " + n),
  h[n] = e
```

根据 App.pages 的配置验证有效性

把页面**选项对象(obj)** 存放在一个Map中

小程序如何创建页面实例

页面创建：拷贝页面选项对象(obj)，创建新的页面实例对象

代码来自开发者工具

```
_ = function(e, t, n) {
var r = void 0;
h.hasOwnProperty(e) ? r = h[e] : ((0,
c.warn)("Page route 错误", "Page[" + e + "] not found.
r = {}),
b.newPageTime = Date.now();
var o = new u.default(r,t,e);
w(o, t),
```

1. 以**跳转路径作为 key** 获取选项对象

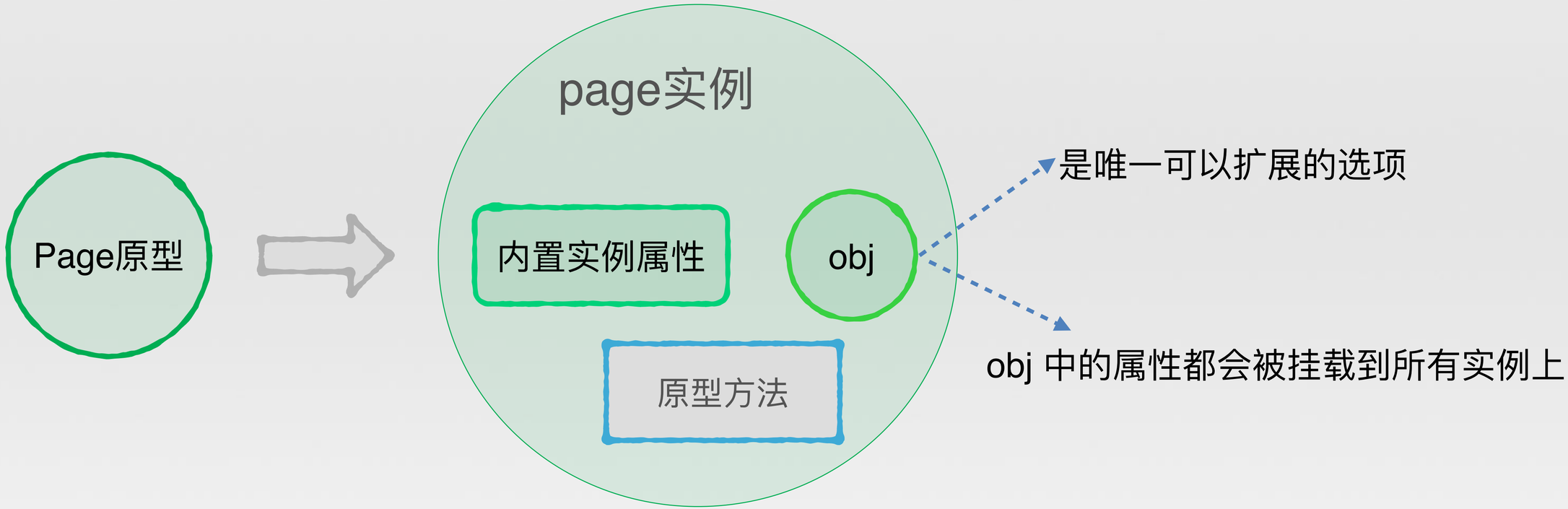
2. **实例化**一个页面对象，并将选项对象的全部属性拷贝至该实例上

小程序如何创建页面实例

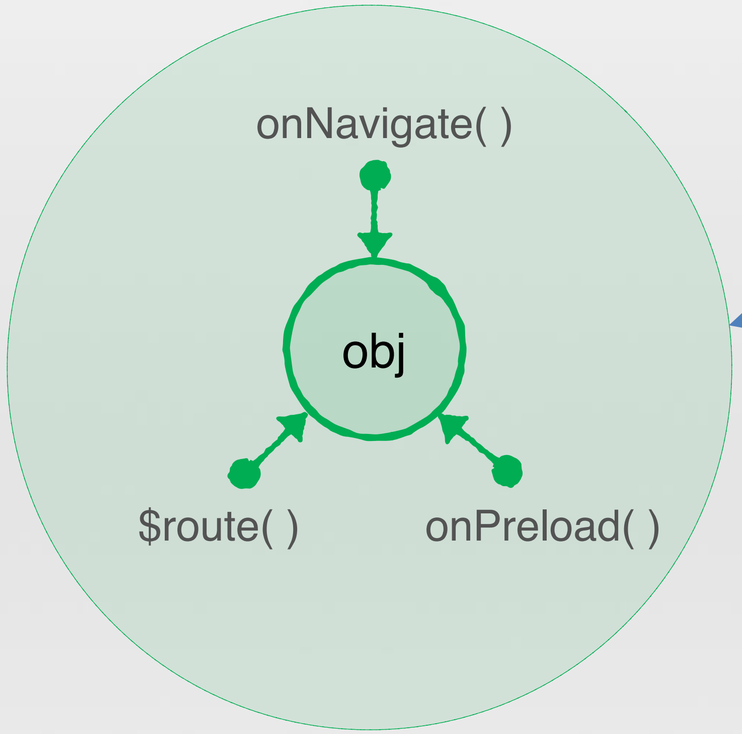
页面创建：拷贝页面选项对象(obj)，创建新的页面实例对象

小程序如何创建页面实例

页面创建：拷贝页面选项对象(obj)，创建新的页面实例对象



扩展页面的生命周期



对小程序的Page方法进行封装，以便**装饰**页面选项对象

```
function P(name, option) {

    // mixin component defs
    Component.use(option, option.comps, `Page[${name}]`)

    if (option.onNavigate){
        let onNavigateHandler = function (url, query) {
            option.onNavigate({url, query})
        }
        console.log(`Page[${name}] define "onNavigate".`)
        dispatcher.on('navigateTo:'+name, onNavigateHandler)
```

监听对应的跳转事件，以调用onNavigate方法

注意：扩展的方法可能在页面实例化前执行，所以无法访问实例属性，例如：onNavigate

强化 onNavigate( )

50 ~ 300 ms

App.onLaunch( )  Page.onLoad( )

利用程序启动到Page实例化的时间差，触发 onNavigate 预先加载页面数据

从小程序1.1.0开始，onLaunch方法可以获取将要访问页面 path 与及 query，所以可以预知将加载的页面

扩展的原型方法与实例方法的区别

```
Page({
  count : 0,
  onLoad: function(options) {
    console.log(++this.count);  //永远等于1
  }
}
```

this指的是新创建的实例，每次都不一样

```
Page({
  count : 0,
  onNavigate: function(options) {
    console.log(++this.count);  //每次+1
  }
}
```

this 指的是页面选项对象，每次指的是同一个

$route( ) 路由函数实现原理

1. 约定规则，根据页面地址找到对应的页面选项对象

2. 调用选项对象的 onNavigate 方法；如果没有定义，则跳过

4. 调用小程序接口方法 wx.navigateTo 执行真正的页面跳转

… 实现跳转截流，**防止用户重复点击**

预加载下一访问页面的数据

预加载下一访问页面的数据

< 以数据为根据的行为预判 />

首页

频道页(第二个页卡)

通过数据分析，发现进入首页的用户有50%的概率会访问第二个页卡

预加载第二个页卡的数据，
可以提高用户访问速度

在首页空闲后，预加载频道页数据

```
onReady: function(){
    //预加载频道页
    this.$preLoad("/pages/channel/index")
}
```

$preload 触发目标页面的 onPreload 方法

增加 onPreload 生命周期函数

```
onPreload: function(query){
    this.$cache('channel', this.fetchData(query));
}
onLoad: function(query) {
    this.$cache('channel') || this.fetchData(query);
}
```

注意：预请求数据缓存在Storage中，**避免占用过多内存**只能实现preload，不能实现prerender

在首页空闲后，预加载频道页数据

```
onReady: function(){
    //预加载频道页
    this.$preLoad("/pages/channel/index")
}
```

$preload 触发目标页面的 onPreload 方法

增加 onPreload 生命周期函数

```
onPreload: function(query){
    this.$cache('channel', this.fetchData(query));
}
onLoad: function(query) {
    this.$cache('channel') || this.fetchData(query);
}
```

为什么不先展示历史数据，再刷新?

注意：预请求数据缓存在Storage中，避免占用过多内存只能实现preload，不能实现prerender

为什么不先展示历史数据，再刷新?

1. 异步刷新的方式会增加渲染消耗，延长关键内容呈现时间

2. 大数据更新的传输消耗较大，小程序需把data转成字符串传给UI线程

✘

```
data: {
    list: ['a','b','c', ...]
},
onClick: function () {
    this.list.forEach(function () {
        ...
    })
}
```

```
data: {},
onLoad: function () {
    this._list = ['a','b','c', ...]
},
onClick: function () {
    this._list.forEach(function () {
        ...
    })
}
```

✔

优化Tips：非渲染用数据不在挂在data上，可挂载在页面实例上供内部逻辑使用

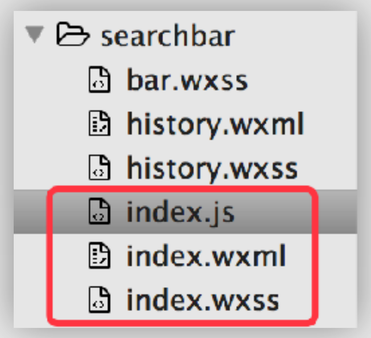
不依赖任何工具的组件化

< 提高项目的可维护性 />

组件的定义

组件的定义与Page类似，包含：js，wxml，wxss

目录结构



定义组件原型

```
module.exports = C('searchbar', function(){
    return {
        data: {...
        },
        onLoad: function(){...
        },
        /**
         * 搜索界面出现的时候，steps:
         * 1. 加载历史记录
         */
        onShow: function () {...
        },
```

C 是组件定义方法

使用组件

导入组件原型

```
module.exports = P('search', {
    {
        data: {...
        },
        comps: [
            require('../../comps/searchbar/index.js')()
        ],
        onLoad: function (query) {...
        },
```

导入组件样式

```
/* tabbar */
@import "/comps/searchbar/index.wxss";
@import "/comps/works/works.wxss";
@import "./result/correct.wxss";
@import "./result/com.wxss";
@import "./result/empty.wxss";

.bg_con {
  position: absolute;
```

导入组件模板

```
<import src="/comps/searchbar/index.wxml"/>
<view class="search page flb-vt">
    <!-- 搜索输入栏 -->
    <template is="searchbar" data="{{...searchbar}}"/>
```

P 是上文提及的对Pages的封装方法
约定 comps 属性作为挂载子组件的入口

子组件的属性会被合并到页面选项对象上

```
C('comp', {
    data: {
        name: "comp"
    },
    onLoad: function () {
        console.log("I am a component");
    },
    onCompTap: function() {

    }
})
```

+

```
//页面定义
{
  data: {
      name : "I am a page"
  },
  onLoad: function(query) {
    console.log("I am a page");
  }
}
```

=

```
//bang!
{
  data: {
      name : "I am a page",
      comp: {
          name : "comp"
      }
  },
  onLoad: function(query) {
    comp.onLoad();  //I am a component
    page.onLoad();  //I am a page
  },
  onCompTap : function(){
  }
}
```

总结

Page 的封装方法 **P**

实例方法

- 自定义路由方法：$route()
- 预加载方法：$preload()
- 公共数据存储： $put(), $take()
- 使用 LS 实现的 session 缓存：$cache()

扩展页面生命周期

- onNavigate
- onPreload

组件化

- 组件定义方法
- comps 入口

延伸扩展

实例方法

- 防重复跳转的代理方法: $bindRoute()
- 组件间的通信：$on, $emit, $off

THANKS

关开设