

【数据层】引用组件实现

实现核心

- 关联关系 【这个是最重要的】
- 数据来源
- 联动效果：通过代理拦截setter & 关联id列表实现
- 属性变量控制：控制更改

核心功能

- 创建
- 删除
- 恢复
- 推送
- 重置
- 分离
- 替换

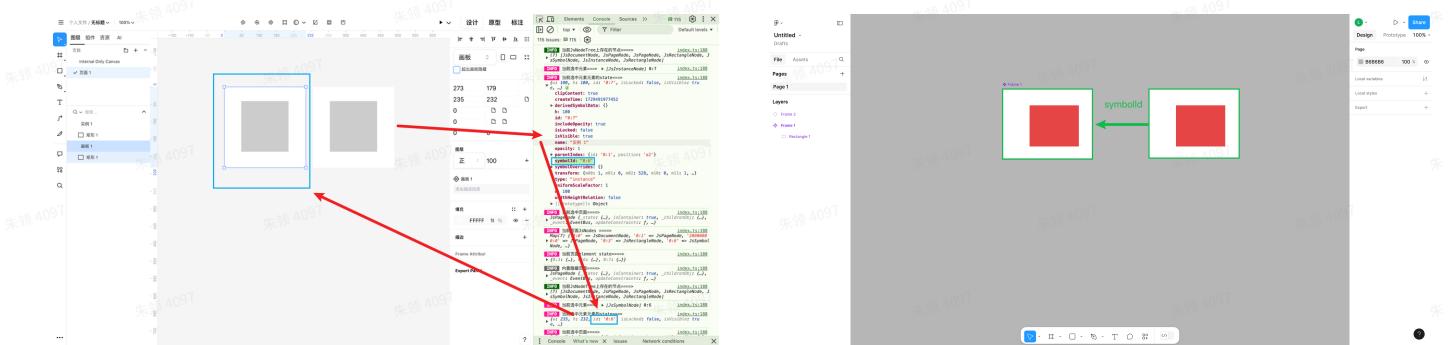
关联关系整理



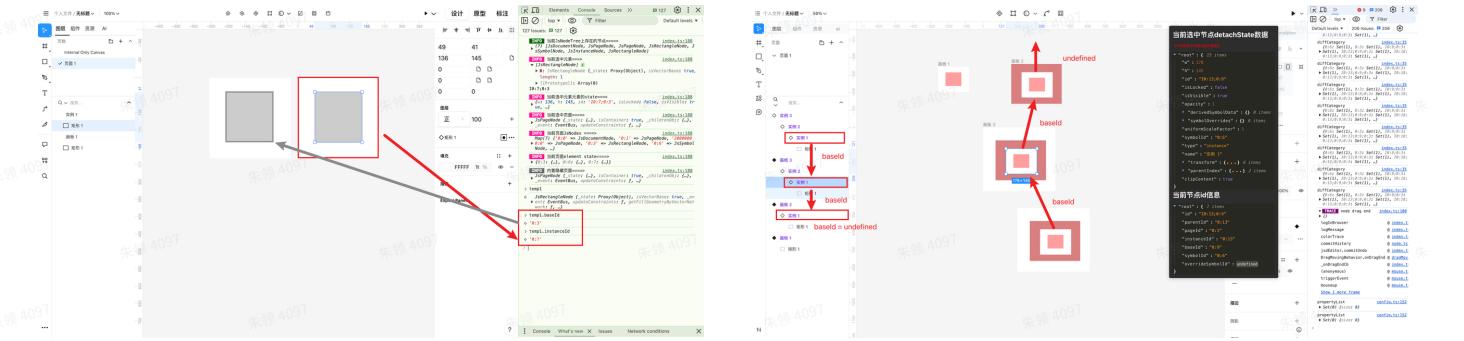
当前节点id信息

```
▼ "root": { 9 items
  "id": "0:12"
  "parentId": "0:1"
  "pageId": "0:1"
  "instanceId": undefined
  "baseId": undefined
  "symbolId": undefined
  "symbolMasterId": ""
  "overrideSymbolId": undefined
  ▼ "symbolRelationSet": [ 2 items
    0: "0:13"
    1: "0:14"
  ]
}
```

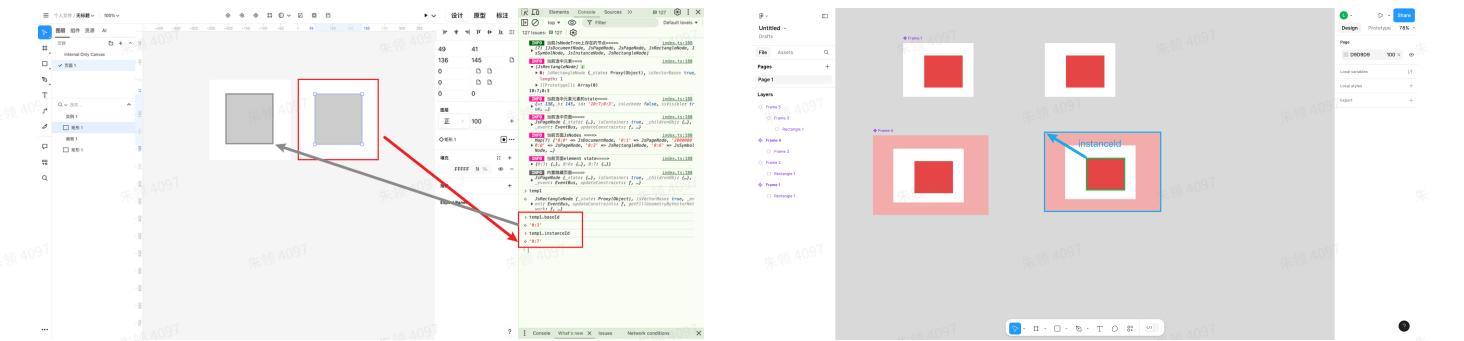
- symbolId: instance指向symbol节点的关联，这个是会存到state中的



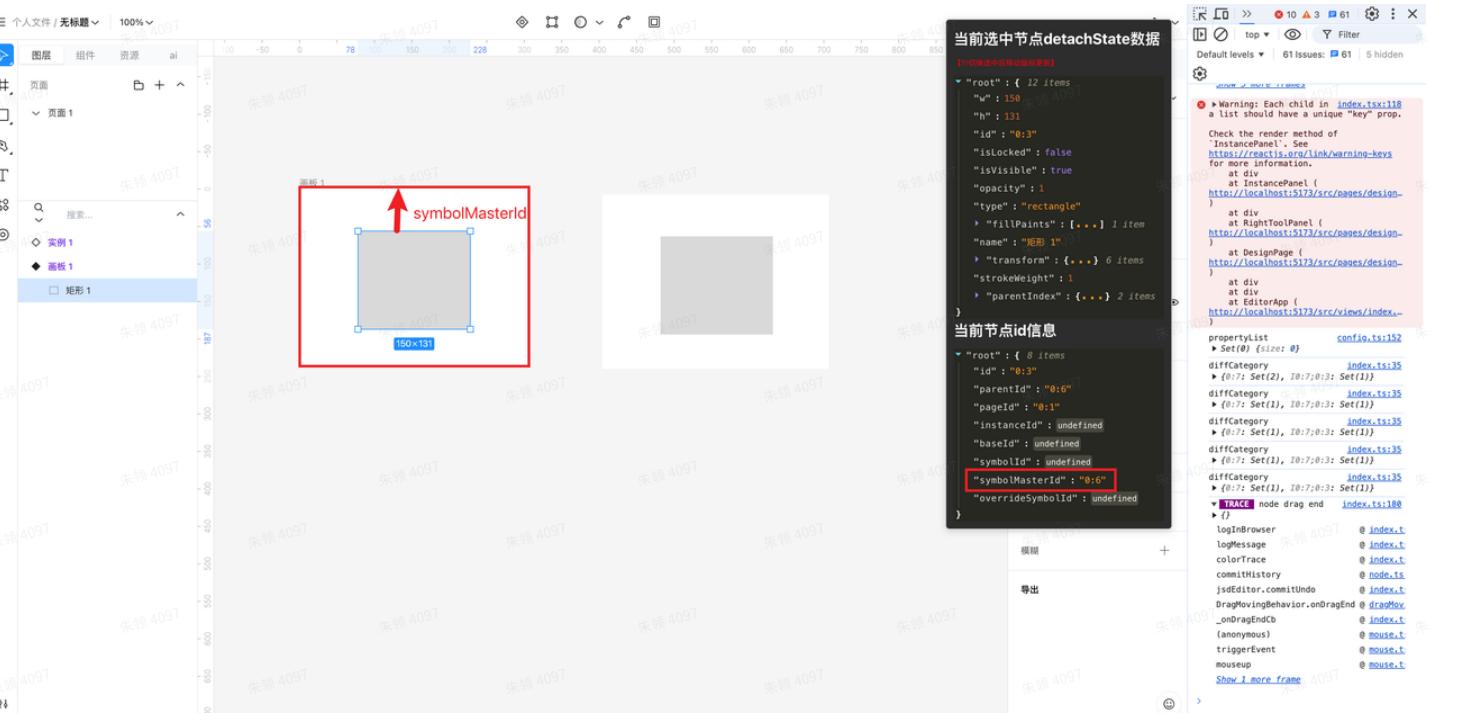
- baseId: symbol子集的id，这个是临时jsNode上的数据，标识symbol子集的关联关系



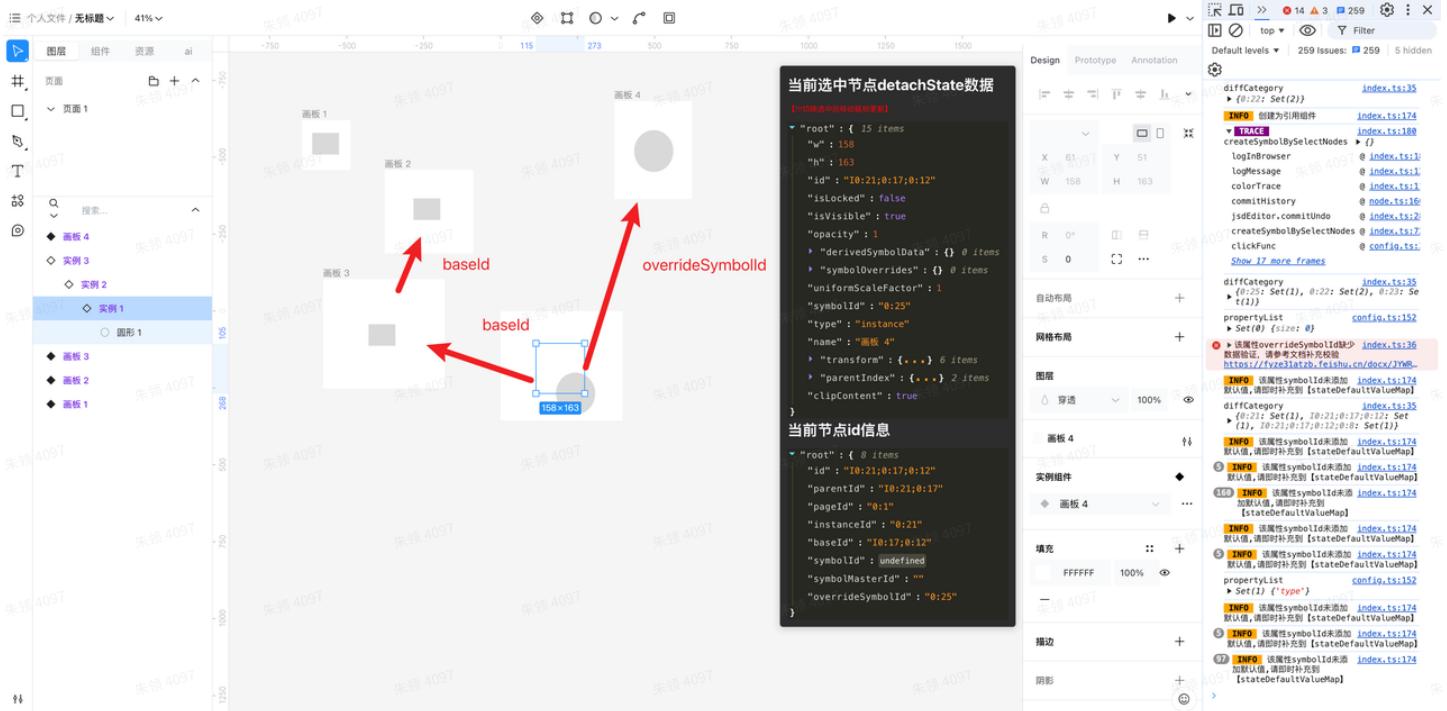
- **instanceId:** 判断是否是instanceIdchild, instanceIdchild跟最外层instance的关系, 这个是临时jsNode上的数据



- **symbolMasterId:** symbolChild & symbol 的关系用symbolMasterId, 这个是临时jsNode上的数据



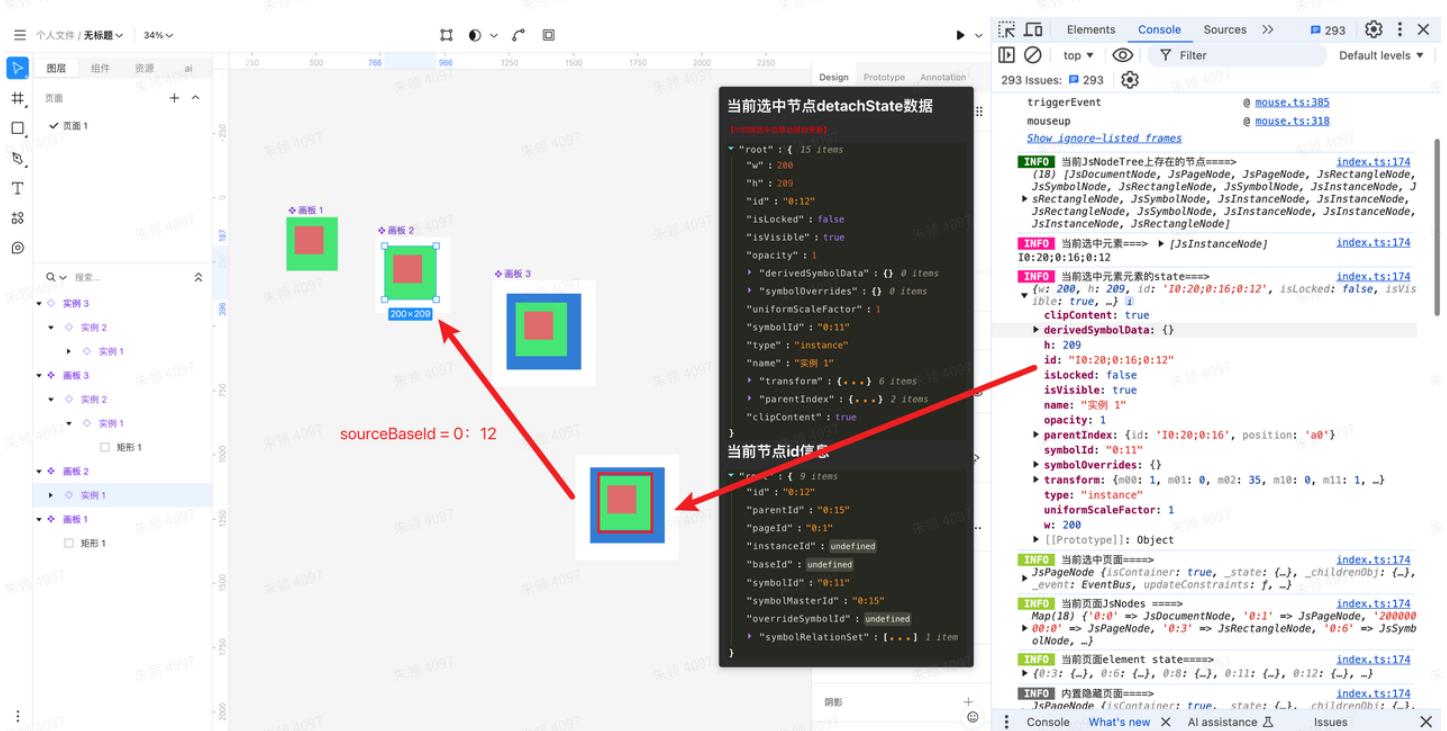
- **overrrsSymbolId:** 替换组件用overrrsSymbolId, 用于指向替换后的symbol, 替换child类型instance时overrrsSymbolId是放置在override中的, 替换最外层instance则直接替换symbolId



- symbolRelationSet: 与symbol关联的instance的id列表

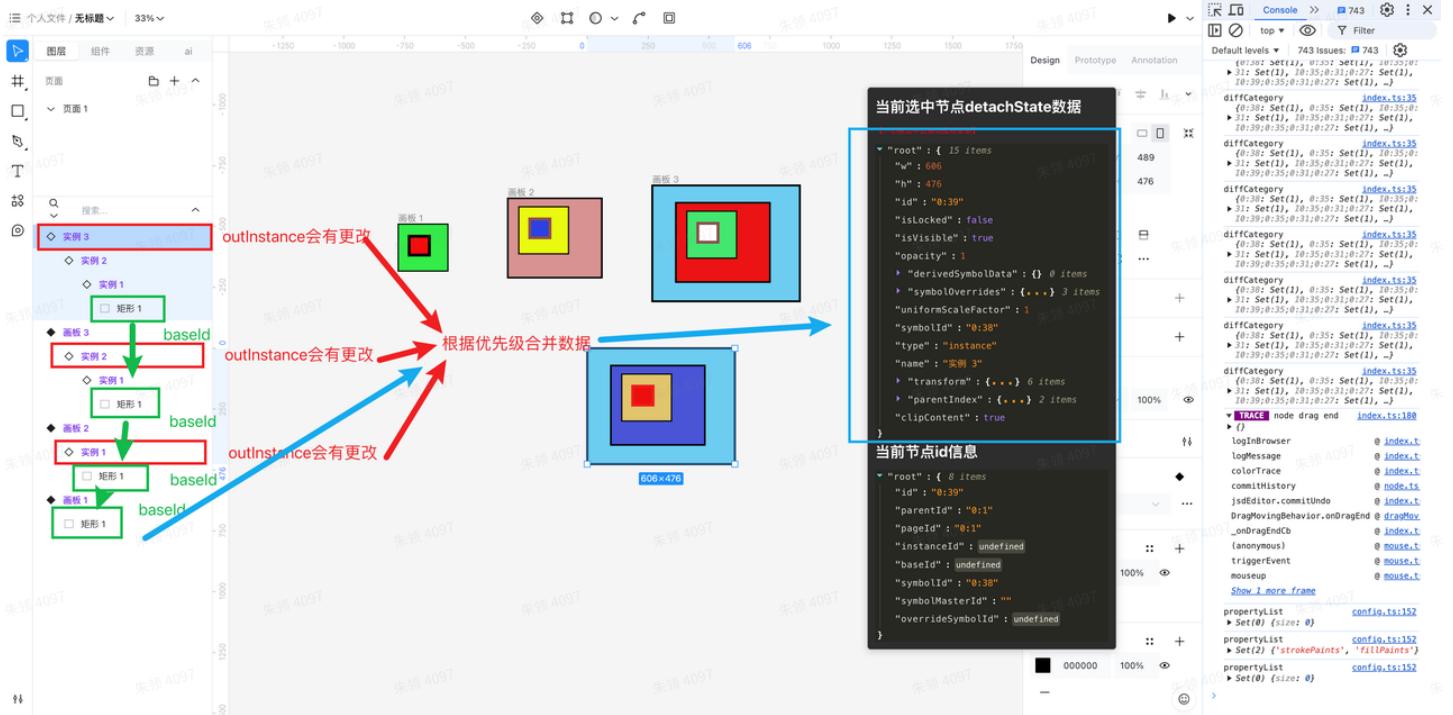
几个核心node

sourceBaseNode: 最原始的baseId对应的节点



instanceChild数据来源优先级

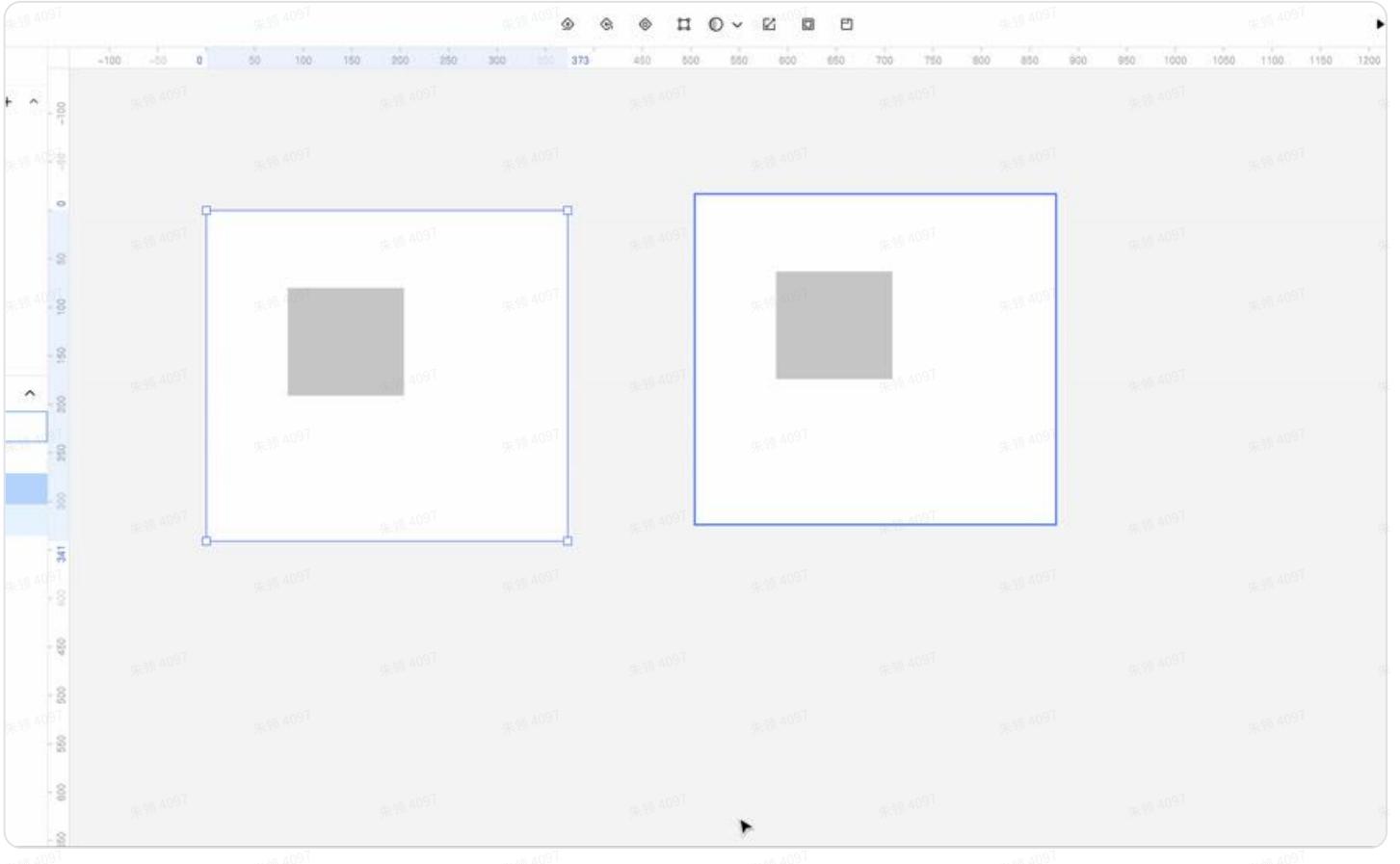
1 最外层instance上的overrides上的数据 > baseNode上的overrides数据 > symbol上的数据



联动效果实现原理: SymbolRelationManager ✅

这个关联列表可能变更的情况是: symbolId变化, 或者overrideSymbolId变化

- 1 通知symbol 关联组件: updateSymbolRelation, 通知根据类型更新
- 2 生成关联列表: generateSymbolRelation



更新通知

```
export class DataDecoratorManager implements IDataDecoratorManager {  
    nodePropertyDecorator<T extends Function>(internalFunc: T, node: IJsBaseNode, category: Category, prop:  
        if (this.isFrozen) {  
            return;  
        }  
        if (!dataValidator.validate(propertyKey, newValue)) {  
            return;  
        }  
        this.collector.collectValue(node.id, { key: propertyKey, value: newValue });  
        internalFunc.call(this, newValue);  
        this.dataEmit.generateEmitKeys(node.id, category);  
  
    stateAddDecorator<T extends Function>(internalFunc: T, state: IBaseState) {  
        if (this.isFrozen) {  
            return;  
        }  
        this.collector.collectValue(state.id, { value: state });  
        const result = internalFunc.apply(this, [state]);  
        this.dataEmit.generateEmitKeys(state.id, 'add');  
        return result;  
    }  
  
    stateRemoveDecorator<T extends Function>(internalFunc: T, id: Id) {  
        if (this.isFrozen) {  
            return;  
        }  
        this.collector.collectValue(id, {});  
        const result = internalFunc.apply(this, [id]);  
        this.dataEmit.generateEmitKeys(id, 'remove');  
        return result;  
    }  
}
```

数据存储：对于instance的代理generateInstanceState

数据一部分存储在本身的overrides中更新&获取同时代理

```
const generateInstanceState = function (instanceState: IInstanceState) { > getOverrideValue Aa ab .*
```

```
//更换覆盖属性 代理实现State 代理
const state = new Proxy(instanceState, {
  get(target, prop: string, receiver) {
    const { symbolOverrides, derivedSymbolData } = instanceState;
    const overrideValue = symbolOverrides[instanceState.id] ?? {};
    const derivedValue = derivedSymbolData[instanceState.id] ?? {};
    const symbolData = getStateById(instanceState.symbolId) as ISymbolState;
    if (prop in overrideValue) {
      return overrideValue[prop as OverrideKey];
    }

    if (prop in derivedValue) {
      return derivedValue[prop as DerivedKey];
    }
  },
  王伟, 2个月前 · feat(jsd-data): 注册节点BasicState 数据
  set(target, prop: OverrideKey | DerivedKey, value: OverrideValue[OverrideKey]) {
    const instance = JsNodes.getNodeById(instanceState.id) as IJsInstanceNode;

    if (prop in target) {
      Reflect.set(target, prop, value);
    } else {
      instance.updateSymbolOverrides(instance.id, prop as OverrideKey, value);
    }
    return true;
  }
});
```

数据存储：对于instanceChild的代理generateInstanceState

对于instanceChild，它没有实体数据，他的数据存储在最外层的instance的overrides中

```
const generateInstanceChildState = function (instance: IJsInstanceNode, baseNode: IJsContainerNode) {
  const overrideSymbolId = ownState['overrideSymbolId'];

  if (overrideSymbolId) {
    const symbolState = getStateById(overrideSymbolId);
    state = symbolState;
  }

  // 更换覆盖属性 代理实现State 代理
  const childState = new Proxy(state, {
    get(target, prop: string, receiver) {
      if (prop in ownState) {
        return ownState[prop as keyof InstanceChildOwnState];
      }
      const overrideSymbolId = ownState.overrideSymbolId;
      if (overrideSymbolId && derivedKeys.includes(prop as DerivedKey)) {
        const baseId = ownState.baseId;
        const baseState = JsNodes.getNodeById(baseId)!.state;
        return baseState[prop as DerivedKey];
      }
      return Reflect.get(target, prop, receiver);
    },
    set(target, prop, value) {
      if (derivedKeys.includes(prop as DerivedKey)) {
        instance.updateDerivedSymbolData[ownState.id, prop as DerivedKey, value as DerivedValue[DerivedKey]];
      }
      instance.updateSymbolOverrides(ownState.id, prop as OverrideKey, value as OverrideValue[OverrideKey]);
      ownState[prop as DerivedKey | OverrideKey] = value;
      return true;
    },
    deleteProperty(target, prop) {
      if (prop in target) {
        delete ownState[prop as keyof InstanceChildOwnState]; // 实际删除
        return true; // 表示删除成功
      }
    }
  });
  return childState;
}
```

无实体数据的instanceChild是如何映射的

当构造instance时，找到symbol，遍历symbol的结构映射出instance的结构

```
1 // 递归处理子集
2 (function deepHandle(baseNode: IJsContainerNode, parent: IJsContainerNode) {
3   baseNode.children.forEach((child) => {
4     const id = generateInstanceId(parent, child);
5     // 处理替换逻辑
6     if (isInstanceNode(child)) {
7       const overrideSymbolId = instance.getOverrideValue(id)[
8         'overrideSymbolId'];
9       const overrideSymbolNode = self.getNodeById(overrideSymbolId!) as
10      IJsSymbolNode;
11      if (overrideSymbolNode) {
12        const instanceChild = self.constructInstanceChild(instance, child,
13          parent);
14        if (isContainerNode(child)) {
15          deepHandle(overrideSymbolNode, instanceChild as
IJsContainerNode);
16        }
17        return true;
18      }
19    }
20  });
21}
```

```
16         }
17         const instanceChild = self.constructInstanceChild(instance, child,
18         parent);
18         // 递归创建其他节点
19         if (isContainerNode(child)) {
20             deepHandle(child, instanceChild as IJsContainerNode);
21         }
22     );
23 }(symbol, nearestInstance));
```

```
1 static constructInstanceChild(instance: IJsInstanceNode, baseNode:
2 IJsBaseNode, parent: IJsContainerNode) {
3     const childState = generateInstanceState(instance, baseNode, parent);
4     // 构造节点
5     const childNode = this.constructNode(childState, true) as IJsBaseNode;
6     if (childNode) {
7         // 重新构建父子级关系
8         parent.appendChildNode(childNode);
9     }
10 }
```

Proxy在引用组件中的作用

- 代理具有实体数据的instance的state: generateInstanceState
- 代理instanceChild的state: generateInstanceState

generateInstanceState

数据来源: overrides + 自身属性

注意: 数据更新存储都是在自身代理上处理

```
if (!thisJsNode || force) {
    if (elementState.type === 'instance' && !elementState.instanceId) {
        elementState = generateInstanceState(elementState as IInstanceState);
    }
    thisJsNode = State2JsNodeService.transform(elementState);
}
```

```

// 解决循环引用先放到这里
const generateInstanceState = function (instanceState: IInstanceState) {
  // 更换覆盖属性 代理实现State 代理
  const state = new Proxy(instanceState, {
    get(target, prop: string, receiver) {
      const { symbolOverrides, derivedSymbolData } = instanceState;
      const overrideValue = symbolOverrides[instanceState.id] ?? {};
      const derivedValue = derivedSymbolData[instanceState.id] ?? {};
      const symbolData = getStateById(instanceState.symbolId) as ISymbolState;
      if (prop in overrideValue) {
        return overrideValue[prop as OverrideKey];
      }

      if (prop in derivedValue) {
        return derivedValue[prop as DerivedKey];
      }

      if (prop in target) {
        return Reflect.get(target, prop, receiver);
      }

      return symbolData[prop as keyof ISymbolState];
    },
    set(target, prop: OverrideKey | DerivedKey, value: OverrideValue[OverrideKey]) {
      const instance = JsNodes.getNodeById(instanceState.id) as IJsInstanceNode;

      if (prop in target) {
        Reflect.set(target, prop, value);
      } else {
        instance.updateSymbolOverrides(instance.id, prop as OverrideKey, value);
      }
      return true;
    }
  });
  return state;
};

```

generateInstanceState

数据来源：最外层instance的overrides中的属性 + baseNode的outInstance的overrides中的属性 + baseNode自身的属性

注意：数据更新存储都是在outInstance上处理，instanceChild本身无实体数据

```

1
2 const generateInstanceState = function (instance: IJsInstanceNode,
  baseNode: IJsBaseNode, parent: IJsContainerNode) {
3   // 代理baseNode的state和outInstance上overrides中的数据
4   let state = baseNode.state as IBaseState;
5   const id = generateInstanceChildId(parent, baseNode);

```

```
6 // 判断是不是 Instance 并且是否存在过替换组件
7 // 替换过的组件 位置尺寸属性 type 在 Base中 获取，其他属性在symbol中的state中获取
8 const overrideValue = instance.getOverrideValue(id);
9 const derivedValue = instance.getDerivedValue(id);
10
11 // 这样将overrides聚合是因为避免多层嵌套Proxy导致性能问题
12 const ownState: InstanceChildOwnState = {
13   baseId: state.id,
14   id: id,
15   instanceId: instance.id,
16   parentIndex: {
17     id: parent.id,
18     position: state.parentIndex.position
19   },
20   ...derivedValue,
21   ...overrideValue,
22 };
23
24 // 这是一个业务要求，baseNode的锁定和显示隐藏要通信
25 if (state.type === 'instance') {
26   ownState.isLocked = state.isLocked;
27   ownState.isVisible = state.isVisible;
28 }
29
30 // 如果Instance 发生了替换
31 const overrideSymbolId = ownState['overrideSymbolId'];
32
33 // 发生替换后只有w,h,transfrom走baseNode, 其他的走替换后的symbol
34 if (overrideSymbolId) {
35   if ((state as IInstanceState).symbolId) {
36     ownState.symbolId = (state as IInstanceState).symbolId;
37   }
38   const symbolState = getStateById(overrideSymbolId!);
39   state = symbolState;
40 }
41
42 // 更换覆盖属性 代理实现state 代理
43 const childState = new Proxy(state, {
44   get(target, prop: string, receiver) {
45
46     if (prop in ownState) {
47       if (prop === 'parentIndex') {
48         const parentIndex = ownState[prop as keyof InstanceChildOwnState] as
49           ParentIndex;
50         parentIndex.position = state.parentIndex.position;
51         return parentIndex;
52     }
53   }
54 }
```

```

52         return ownState[prop as keyof InstanceChildOwnState];
53     }
54     const overrideSymbolId = ownState.overrideSymbolId;
55     if (overrideSymbolId && derivedKeys.includes(prop as DerivedKey)) {
56         const baseId = ownState.baseId;
57         const baseState = JsNodes.getNodeById(baseId)!.state;
58         return baseState[prop as DerivedKey];
59     }
60     return Reflect.get(target, prop, receiver);
61 },
62 set(target, prop, value) {
63     if (derivedKeys.includes(prop as DerivedKey)) {
64         instance.updateDerivedSymbolData(ownState.id, prop as DerivedKey,
65         value as DerivedValue[DerivedKey]);
66         instance.updateSymbolOverrides(ownState.id, prop as OverrideKey, value as
67         OverrideValue[OverrideKey]);
68         ownState[prop as DerivedKey | OverrideKey] = value;
69         return true;
70     },
71     deleteProperty(target, prop) {
72         if (prop in ownState) {
73             delete ownState[prop as keyof InstanceChildOwnState]; // 实际删除
74             return true; // 表示删除成功
75         } else {
76             return true; // 表示删除失败
77         }
78     });
79 }
80 return childState;
81 };

```

数据通信

symbol改动是如何通信instance的

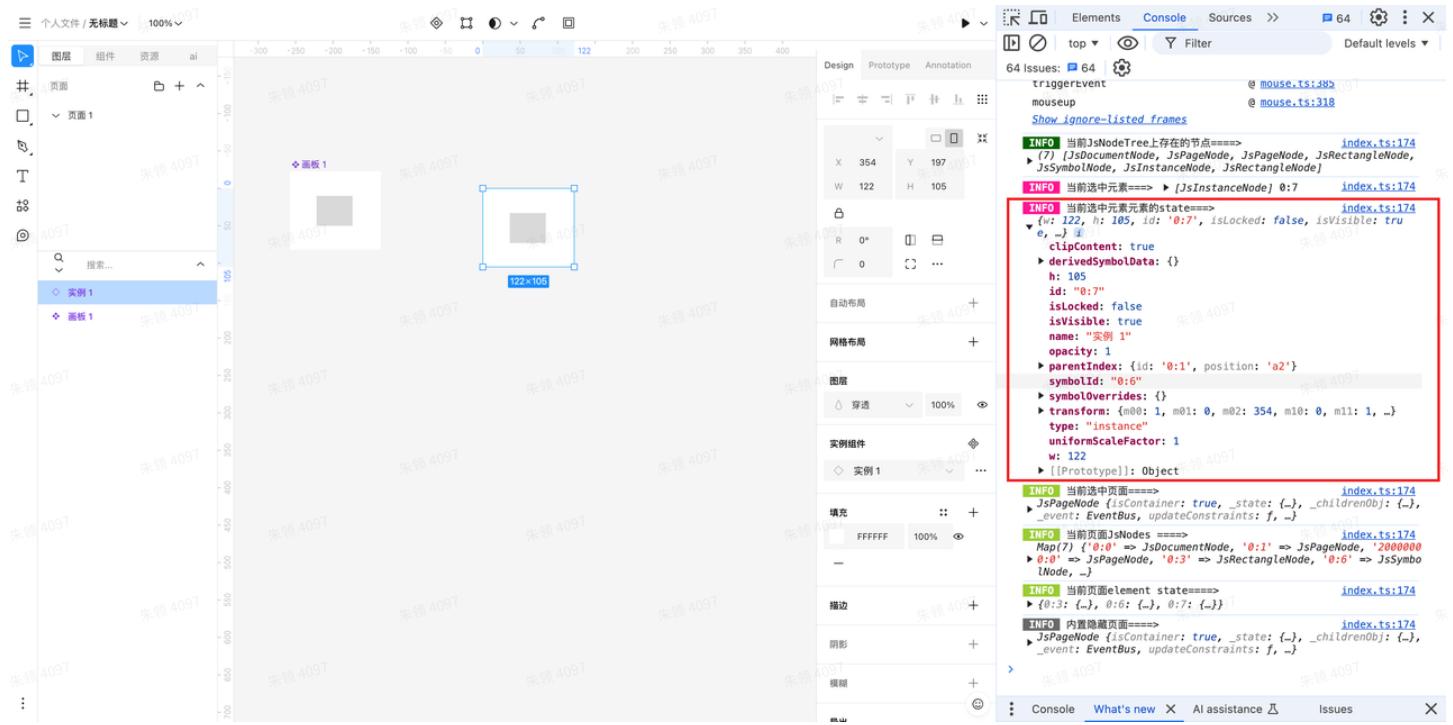
主要是在构建instance节点时，代理instance其本身的数据，instance是具备实体state数据的
getter拦截：

- 先从overrides查找
- 再从本身state查找
- 最后从symbol上查找

setter拦截：

- state中有的字段直接更新到state上
- state中没有的放置到overrides中

这样在JsNodeTree渲染获取数据时就达到了联动效果



创建

创建symbol

- 仅frame能传入创建
- 转换frame为symbol
- 将frame的子级的父级修改为新的symbol

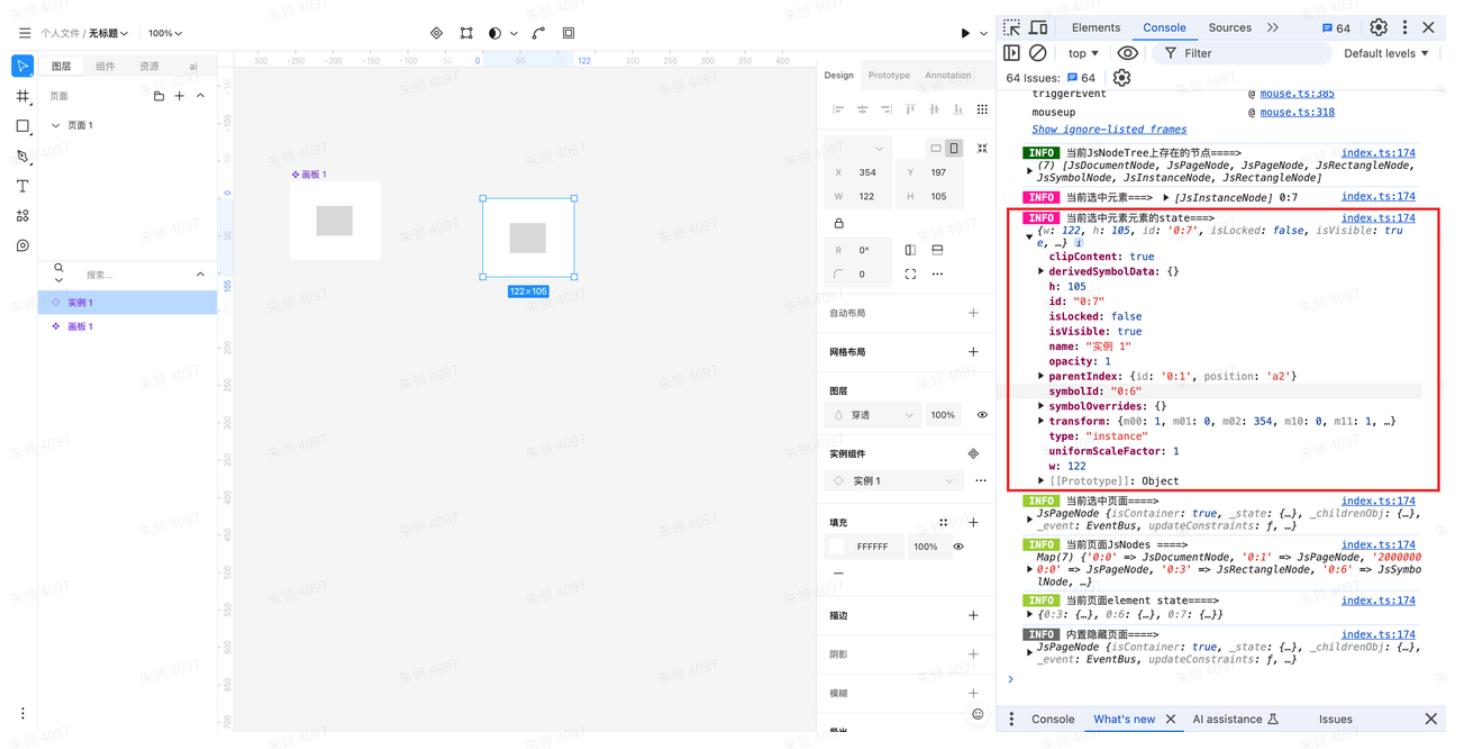
```

1  /**
2   * 创建引用组件
3   * @param id
4   * @returns
5  */
6  export const createSymbolNode = (frameNode: IJsFrameNode): IJsSymbolNode | undefined => {
7    if (!frameNode) {
8      return;
9    }
10   if (frameNode.type !== 'frame') {
11     logHelper.error('传入节点存在非frame，仅frame可创建为symbol');
12     return;
13   }
14   const basicSymbol = getBasicSymbol();
15   const oldStateData = frameNode.getState();
16   // 构建新结构
17   const newSymbolData = {
18     ...basicSymbol,
19     ...oldStateData,
20     id: generateIncID(EIncIdType.Element),
21     type: 'symbol'
22   } as ISymbolState;
23   // 增加新节点
24   if (oldStateData) {
25     JsNodes.addNode(newSymbolData);
26   }
27   // 构造新节点
28   const symbolNode = JsNodes.getNodeById(newSymbolData.id) as IJsSymbolNode;
29   frameNode.children.forEach((node) => {
30     if (!node) {
31       return;
32     }
33     node.changeParentIndex(symbolNode.id);
34   });
35   // 移除旧节点
36   frameNode.removeNode();
37   return symbolNode;
38 };

```

创建instance

核心数据，用于承载变更，和关联关系



```
1 derivedSymbolData: {},
2 symbolOverrides: {},
3 symbolId: symbolId,
4
5 /**
6 * 基础矩形默认schema数据
7 * @returns {IFrameState}
8 */
9 export function getBasicInstance(symbolId: Id): IInstanceState {
10   const symbolNode = JsNodes.getNodeById(symbolId)!;
11   if (!symbolNode) {
12     logHelper.warn('symbolNode节点不存在');
13   }
14   return {
15     w: symbolNode.w,
16     h: symbolNode.h,
17     ...getDefaultBaseSchema(),
18     derivedSymbolData: {},
19     symbolOverrides: {},
20     uniformScaleFactor: 1,
21     symbolId: symbolId,
22     type: 'instance',
23     name: generateDefaultLayerName('instance'),
```

```
24     transform: defaultTransform({ x: symbolNode.x, y: symbolNode.y }),  
25     parentIndex: generateParentIndex(),  
26     clipContent: true,  
27   };  
28 }
```

```
1 export function createInstance(symbolId: Id) {  
2   const instanceState = getBasicInstanceState(symbolId);  
3   JsNodes.addNode(instanceState);  
4   const symbol = JsNodes.getNodeById(symbolId)!;  
5   const instance = JsNodes.getNodeById(instanceState.id)!;  
6   instance.changePos({ x: symbol.x + symbol.w + 20, y: symbol.y });  
7   return instance;  
8 }
```

创建instanceChild

instanceChild无实际state数据

- instanceChild无实际数据，是根据instance和symbol映射出临时的jsNodes的

```
/**  
 * 根据instance构造 child 节点  
 * @param {IJsInstanceNode} instance - 节点  
 */  
static constructInstanceChildren(instance: IJsInstanceNode) {  
  const self = this;  
  const symbol = this.getNodeById(instance.symbolId) as IJsSymbolNode;  
  
  // 递归处理子集  
  (function deepHandle(baseNode: IJsContainerNode, parent: IJsContainerNode) {  
    baseNode.children.forEach((child) => {  
      let overrideSymbolNode;  
      if (isInstanceNode(child)) {  
        const id = generateInstanceChildIndex(parent, child);  
        const overrideSymbolId = instance.getChildOverrideVal(id, 'symbolId');  
        overrideSymbolNode = self.getNodeById(overrideSymbolId!) as IJsSymbolNode;  
      }  
      const instanceChild = self.constructInstanceChild(instance, child, parent, overrideSymbolNode);  
      if (i.IContainerNode(child!) && instanceChild) {  
        deepHandle(overrideSymbolNode as IJsContainerNode ?? child, instanceChild as IJsContainerNode);  
      }  
    })  
  })(symbol, instance);  
};
```

```

2     * 根据Id构造指定节点
3     * @param {string} id - 节点ID
4     * @returns {Node} - 返回构建的节点
5   */
6   static constructNode(elementState: IBaseState, force: boolean = false) {
7     const elementId = elementState.id;
8     let thisJsNode = this.nodes.get(elementId);
9     if (!thisJsNode || force) {
10       thisJsNode = State2JsNodeService.transform(elementState);
11     }
12
13     if (thisJsNode) {
14       this.nodes.set(elementId, thisJsNode);
15       const parent = thisJsNode.parent as IJsContainerNode;
16       if (parent) {
17         parent.addChildNode(thisJsNode);
18       }
19       // 构造!!!
20       if (isInstanceNode(thisJsNode) && !isInstanceChildNode(thisJsNode)) {
21         this.constructInstanceChildren(thisJsNode);
22       }
23       // 节点创建时调用触发
24       thisJsNode.onCreate?.();
25     }
26     return thisJsNode;
27   }

```

instanceChild的state数据来源于

- instanceChild的state数据来源于，优先级从上往下，数据层的实现是通过代理baseNode的state去获取instanceChild的state，构造临时的jsNodes
 - 自身的映射关系&父子级关系ownValue
 - 自身更改过的属性overrideVal
 - 'transform', 'w', 'h', 'type' 从baseNode 【baseId指向】 上拿
 - 如果存在替换，样式类属性从替换的symbol节点上拿symbolData
 - 所有其他除以上情况外的数据均从baseNode上拿

```

1 export const generateInstanceChildState = function (instance: IJsInstanceNode,
2   baseNode: IJsBaseNode, parent: IJsContainerNode, symbolNode?: IJsSymbolNode) {
3   let state = baseNode.getState() as IBaseState;
4   const id = generateInstanceId(parent, baseNode);
5   let symbolData = symbolNode?.getState() ?? {};
6
7   // ...
8
9   return state;
10 }

```

```
5 // 判断是不是 Instance 并且是否存在过替换组件
6 // 替换过的组件 位置尺寸属性 type 在 Base中 获取，其他属性在symbol中的state中获取
7
8 const ownValue = {
9   baseId: state.id,
10  id: id,
11  instanceId: instance.id,
12  parentIndex: {
13    id: parent.id,
14    position: state.parentIndex.position
15  }
16 };
17
18 //更换覆盖属性 代理实现State 代理
19 const childState = new Proxy(state, {
20   get(target, prop: string, receiver) {
21     // 优先从 overrideObj 中查找属性
22     const overrideVal = instance.getChildOverrideVal(ownValue.id, prop as
23       OverrideKey | DerivedKey);
24     if (prop in ownValue) {
25       // @ts-ignore
26       return ownValue[prop];
27     }
28     if (overrideVal) {
29       return overrideVal;
30     }
31     if (['transform', 'w', 'h', 'type'].includes(prop)) {
32       return Reflect.get(target, prop, receiver);
33     }
34     // 从替换的组件上拿属性
35     if (prop in symbolData) {
36       // @ts-ignore
37       return symbolData[prop];
38     }
39     return Reflect.get(target, prop, receiver);
40   },
41   set(target, prop: OverrideKey | DerivedKey, value:
42     OverrideValue[OverrideKey]) {
43     // 将newValue直接设置到 overrideObj 中
44     instance.changeChildOverrideVal(ownValue.id, prop, value);
45     return true;
46   }
47 });
48 return childState;
49 };
```

instanceChild的state数据放置于

- 如果是instanceChild会将更改的数据放置在最外层的instance上去，因为instanceChild node 数据是临时的，我们需要持久保存

```
25 export class JsInstanceNode extends JsContainerNode<IJsInstanceNode, IInstanceState> implements IJsInstanceNode {
26     getChildOverrideVal<T extends OverrideKey | DerivedKey>(id: Id, key: T): TypeValueMap<T> {
27         const overrideValue = this.symbolOverrides[id] ?? {};
28         return overrideValue[key] as TypeValueMap<T>;
29     }
30 }
31
32 changeChildOverrideVal<T extends OverrideKey | DerivedKey>(id: Id, key: T, value: TypeValueMap<T>) {
33     // 临时写这几个
34     if(['w', 'h', 'transform'].includes(key)) {
35         this.updateDerivedSymbolData(id, key as DerivedKey, value as TypeValueMap<DerivedKey>);
36     } else {
37         this.updateSymbolOverrides(id, key, value);
38     }
39 }
40
41 updateSymbolOverrides<T extends OverrideKey = OverrideKey>(id: Id, key: T, value: OverrideValue[T]) {
42     if (this.instanceId) {
43         // @ts-ignore
44         super[key] = value;
45         return;
46     }
47     this.changeSymbolOverrides(draft) => {
48         if (!draft[id]) {
49             draft[id] = {};
50         }
51         draft[id][key] = value;
52     };
53 }
54
55 changeSymbolOverrides(recipe: BaseNodeApi.Recipe<SymbolOverrides>) {
56     const symbolOverrides = this.symbolOverrides;
57     if (symbolOverrides) {
58         const newSymbolOverrides = customClone(symbolOverrides, recipe);
59         this.symbolOverrides = newSymbolOverrides;
60     }
61 }
```

derivedSymbolData & symbolOverrides的区别

- derivedSymbolData用于存储instance不可主动更改的数据
- symbolOverrides用于存储instance可主动更改的数据
- 两者不会重合

创建variant

- 只有master能创建变体
- 设置一个变量列表

```
1 export function createVariant(symbolNodes: IJsSymbolNode[]) {
2     const isArraySymbol = symbolNodes.every(symbolNode => symbolNode?.type === 'symbol');
3     if (!isArraySymbol) {
4         logHelper.error('传入节点存在非symbol，仅symbol可创建variant');
5         return;
6     }
7     const symbolIds = symbolNodes.map(symbolNode => symbolNode.id);
```

```
8  const variant = createContainer(symbolIds, 'variant') as IJsVariantNode;
9
10 variant.changeStateGroupPropertyValueOrders((stateGroupPropertyValueOrders)
=> {
11   const newProperty = {
12     property: 'Property 1',
13     values: variant.children.map(item => item.name)
14   };
15   stateGroupPropertyValueOrders.push(newProperty);
16 });
17
18 return variant;
19 }
```

symbol节点 ✅

删除

- symbol在存在关联instance时不会直接删除，而是移至内置页面
- 在每次数据层初始化时，会检测内置page中的symbol，如果没有关联instanc，则删除

```
initClearInternalPage
```

```
1 removeNode() {
2   const parentList = getAllParentIdById(this, true);
3   if (!JsNodeTree.checkNodeIsInInternalPageById(this)) {
4     const internalPageId = JsNodeTree.getInternalPageId();
5     // 记录删除之前的祖先路径
6     this.ancestorPathBeforeDeletion = parentList.map((node) => node.id);
7     super.changeParentIndex(internalPageId);
8   }
9 }
```

描述&link

figma的描述支持富文本，为了避免一直解析富文本字符串，figma用了两个字段去保存描述信息，富文本编辑用description，描述用symbolDescription

```
children: [
  { ..., type: "INSTANCE", name: "21321322" },
  { ...
    guid: { sessionID: 1, localID: 3 },
    phase: "CREATED",
    type: "SYMBOL",
    name: "21321321",
    children: [
      { ..., type: "ROUNDED_RECTANGLE", name: "Rectangle 1" }
    ],
    description: "<p>21321321321</p>",
    userFacingVersion: "7:0",
    isSymbolPublishable: true,
    sharedSymbolVersion: "7:0",
    symbolDescription: "21321321321",
    componentPropDefs: [],
    visible: true,
    opacity: 1,
    size: { x: 246, y: 222 },
    transform: { m00: 1, m01: 0, m02: -410, m10: 0, m11: 1, m12: -256 },
    strokeWeight: 1,
    strokeAlign: "INSIDE",
    strokeJoin: "MITER",
    fillPaints: [
      { ..., type: "SOLID", color: "#000000" }
    ],
    frameMaskDisabled: true,
    symbolLinks: [
      { uri: "https://www.baidu.com" }
    ],
    editInfo: { userId: "1210200424553611378", lastEditedAt: 1730084932, createdAt: 1730084932 }
  }
],
visible: true
```

instance节点

删除 

删除需要维护关联id列表，其他同base

分离 detach 

核心逻辑

- 当前instance & 直接父级instance转换为frame
- 内层instance不转换，但需要变为实体instance，所以任何场景下的分离影响的都是最外层instance的一整个树结构

实现核心

从最外层instance开始遍历，将直接父级instance转换为frame，其他祖先cloneNode，内层instance转为实体instance节点，其他类型子级转为实体节点，构建一颗新的树替换最外层instance的树

```
1 (function traverseChildren(outermostInstance: IJsBaseNode, parent: IJsBaseNode) {
2   outermostInstance.children.forEach((child) => {
3     let curNewNode;
4     if (isInstanceNode(child) && detachInstanceId.includes(child?.id)) {
5       curNewNode = convertInstanceToFrame(child, parent);
6     } else if (isInstanceNode(child)) {
7       curNewNode = child.cloneNode();
8       curNewNode.changeParentIndex(parent.id);
9     } else {
10       const oldData = child.detachState;
11       curNewNode = createNodeByState(child.type, {
12         ...oldData as UnionBaseState,
13         parentIndex: generateParentIndex(parent.id),
14         id: generateIncID(EIncIdType.Element),
15       });
16     }
17     if (detachInstanceId.includes(child.id) || !isInstanceNode(child)) {
18       traverseChildren(child, curNewNode);
19     }
20   });
21 })(outermostInstance, outermostFrame);
```

detachState数据构造【分离核心】

根据数据优先级，合并overrides中的数据：需要收集所有overrides中的数据，如果存在替换需要替换symbolId + baseNode上的数据 = 新节点

- instance类型，到实体instance时仍需遍历，因为最外层instance上的overrides存储更改
- 非instance，收集overrides

恢复restore ✓

根据属性判断是否master放置在internalPage中，恢复

```
1 restore() {
2   const symbol = JsNodes.getNodeById(this.symbolId) as IJsSymbolNode;
3   const originPageId = symbol?.ancestorPathBeforeDeletion as string[];
4   const oldPageParentExist = !!JsNodes.getNodeById(originPageId[0]);
```

```

5     if (!JsNodeTree.checkNodeIsDelete(symbol)) {
6         logHelper.error('symbol节点未在internalPage中');
7         return;
8     }
9     symbol.changeParentIndex(oldPageParentExist ? originPageId[0] :
10        JsNodeTree.getCurrentPageId());
11     symbol.changeAncestorPathBeforeDeletion([]);

```

重置resetInstanceAllChanges

核心：根据更改list，去删除overrides中的覆盖数据，如果存在替换，需要删除替换后的子级，然后重新根据原来的symbol构造回替换前的instance

```

1  /**
2  * @description 重置某个child节点的所有覆盖属性
3  * @param node
4  * @param child
5  * @param property
6  */
7 export const resetChildChange = function (instance: IJsInstanceNode, child:
IBaseNode) {
8     const categoryList = getResetChangePropertyById(instance, child.id);
9     instance.changeSymbolOverrides((draft) => {
10         if (draft[child.id]) {
11             if (draft[child.id].overrideSymbolId) {
12                 (child as IJsInstanceNode).traverseChildren(_child) => {
13                     delete draft[_child.id];
14                     _child.removeNode();
15                 };
16             }
17             delete draft[child.id];
18         }
19     });
20     categoryList.forEach((category) => {
21         internalApi.dataEmit.generateEmitKeys(child.id, category);
22     });
23 };

```

推送pushChangesToMainSymbol

coverOverrideValue ???

替换 swapInstance

核心逻辑

核心是根据替换的instance类型进行处理

1. 标记替换id

- 外层有实体数据的instance: 直接更改symbolId指向, 然后根据symbol的结构重新构建instance结构
- 内层无实体数据的instance: 通过更改overrideSymbolId, overrideSymbolId会放置到最外层instance中, 标记该节点被替换

2. 清理替换之前的覆盖属性, 替换之前的节点的子级可能存在一些更改的覆盖属性

3. 删除替换之前的子级节点

4. 构造新的instance, 其中还需要处理一些业务逻辑

```
1 get overrideSymbolId() {  
2     return this.instance?.getOverrideValue(this.id)['overrideSymbolId']!;  
3 }  
4 @DataTracker('type')  
5 set overrideSymbolId(symbolId: Id) {  
6     if (this.instanceId) {  
7         this.state.overrideSymbolId = symbolId;  
8         generateSymbolRelation(this as IJsBaseNode);  
9     }  
10 }
```

```
1 export const swapInstance = function (currentInstance: IJsInstanceNode,  
2 symbolId: Id) {  
3     const symbolNode = JsNodes.getNodeById(symbolId) as IJsBaseNode;  
4     if (!isSymbolNode(symbolNode)) {  
5         return;  
6     } // 判断嵌套的instance是否符合替换要求  
7     if (!verifyParentCandidate(currentInstance, symbolNode)) {  
8         logHelper.info('不允许替换为其父级组件');  
9         throw Error('不允许替换为其父级组件');  
10    return;  
11 } // 如果是嵌套的instance  
12 const instance = currentInstance.instance ? currentInstance.instance :  
13 currentInstance;
```

```
14 // 获取名字-层级对应的覆盖属性
15 const levelNameOverrideValue = saveLevelNameOverrideValue(currentInstance);
16
17 // 保留缓存属性覆盖属性
18 saveCacheOverrides(currentInstance, instance.symbolOverrides);
19
20 // 覆盖 w h 属性到新的Instance
21 if (!isInstanceChildNode(currentInstance)) {
22     const overridesValues = instance.symbolOverrides[currentInstance.id] ?? {};
23     if (!overridesValues['w']) {
24         instance.stateW = symbolNode.w;
25     }
26     if (!overridesValues['h']) {
27         instance.stateH = symbolNode.h;
28     }
29 }
30
31 // 替换为原来的symbolId 指向时 逻辑处理
32 if (currentInstance.symbolId === symbolId &&
33     currentInstance.overrideSymbolId) {
34     delete currentInstance.state['overrideSymbolId'];
35 } else {
36     if (isInstanceChildNode(currentInstance)) {
37         currentInstance.overrideSymbolId = symbolId;
38     } else {
39         currentInstance.symbolId = symbolId;
40     }
41 }
42
43 // 清理当前子级下所有覆盖属性
44 instance.changeSymbolOverrides((draft) => {
45     currentInstance.traverseChildren((child) => {
46         if (draft[child.id]) {
47             delete draft[child.id];
48         }
49     });
50 });
51
52 // 清理子集node数据
53 currentInstance.traverseChildren((child) => {
54     child.removeNode();
55 });
56
57 // 使用缓存属性到新的Instance
58 coverCacheOverrides(currentInstance, instance);
```

```
60     // 同名同层级覆盖属性逻辑
61     coverLevelNameOverrideValue(currentInstance, instance,
62         levelNameOverrideValue);
63
64     // 构造当前子级
65     constructInstance(currentInstance);
66
67 };
```

overrideSymbolId的利用

主要是在构造的时候判断是否存在替换进而处理替换逻辑，重新构建instance，并且更改数据通信指向

```
/** 根据instance构造 child 节点
 * @param {IJsBaseNode} node - 节点
 */
static constructInstanceChildren(node: IJsBaseNode) {
    const self = this;
    let nearestInstance = isInstanceNode(node) ? node : node.nearestInstance as IJsInstanceNode;
    const instance = node.instanceId ? node.instance! : node as IJsInstanceNode;
    const symbolId = nearestInstance.overrideSymbolId ?? nearestInstance.symbolId;
    const symbol = this.getNodeById(symbolId) as IJsSymbolNode;
    let baseNode = symbol as IJsContainerNode;
    if (isInstanceChildNode(node)) {
        let baseId = node.baseId!;
        if (isInstanceNode(node) && node.overrideSymbolId) {
            baseId = node.overrideSymbolId;
        }
        baseNode = this.getNodeById(baseId) as IJsContainerNode;
    };
    if (!baseNode) {
    };
}
```

instanceChild 【节点，无实际state】

Child数据是如何实现更改的generateInstanceChildState

通过Proxy将更改的数据存储到最外层的instance上去

```
1 export const generateInstanceChildState = function (instance: IJsInstanceNode,
2     baseNode: IJsBaseNode, parent: IJsContainerNode, symbolNode?: IJsSymbolNode) {
3     let state = baseNode.getState() as IBaseState;
4     const id = generateInstanceChildId(parent, baseNode);
5     let symbolData = symbolNode?.getState() ?? {};
6     // 判断是不是 Instance 并且是否存在过替换组件
```

```
7 // 替换过的组件 位置尺寸属性 type 在 Base中 获取，其他属性在symbol中的state中获取
8
9 const ownValue = {
10   baseId: state.id,
11   id: id,
12   instanceId: instance.id,
13   parentIndex: {
14     id: parent.id,
15     position: state.parentIndex.position
16   }
17 };
18 //更换覆盖属性 代理实现State 代理
19 const childState = new Proxy(state, {
20   get(target, prop: string, receiver) {
21     // 优先从 overrideObj 中查找属性
22     const overrideVal = instance.getChildOverrideVal(ownValue.id, prop as
OverrideKey | DerivedKey);
23     if (prop in ownValue) {
24       // @ts-ignore
25       return ownValue[prop];
26     }
27     if (!isUndefined(overrideVal)) {
28       return overrideVal;
29     }
30     if(['transform', 'w', 'h', 'type'].includes(prop)) {
31       return Reflect.get(target, prop, receiver);
32     }
33     // 从替换的组件上拿属性
34
35     if (prop in symbolData) {
36       // @ts-ignore
37       return symbolData[prop];
38     }
39     return Reflect.get(target, prop, receiver);
40   },
41   set(target, prop: OverrideKey | DerivedKey, value:
OverrideValue[OverrideKey]) {
42     // 将newValue直接设置到 overrideObj 中
43     instance.changeChildOverrideVal(ownValue.id, prop, value);
44     return true;
45   }
46 });
47
48 return childState;
49 };
50
```

```
changeChildOverrideVal<T extends OverrideKey | DerivedKey>(id: Id, key: T, value: TypeValueMap<T>) {
    // 临时写这几个
    if (['w', 'h', 'transform'].includes(key)) {
        this.updateDerivedSymbolData(id, key as DerivedKey, value as TypeValueMap<DerivedKey>);
    } else {
        this.updateSymbolOverrides(id, key, value);
    }
}

updateSymbolOverrides<T extends OverrideKey = OverrideKey>(id: Id, key: T, value: OverrideValue[T]) {
    if (this.instanceId) {
        // @ts-ignore
        super[key] = value;
        return;
    }
    [lightbulb icon] 王伟, 2周前 · feat(jsd-data): 替换引用组件
    this.changeSymbolOverrides((draft) => {
        if (!draft[id]) {
            draft[id] = {};
        }
        draft[id][key] = value;
    });
}
```

variant节点

detachState构造

instanceChild

正常情况: baseNode, symboldata, 递归获取最外层overdata

产生替换: 【x,y,w,h,type】 from baseld, 其他来自 replaceSymbol, 递归获取最外层overdata

instance节点

正常情况: baseNode, 最外层overdata, symbolData

产生替换: 【x,y,w,h,type】 from baseld&symbol, 最外层overdata, 其他来自 replaceSymbol

其他节点【不会产生替换】

正常情况: baseNode, 最外层overdata

记录下情况

1. 单独的instance，非instanceChild中的instance，无baseId

直接拿getState即可，数据来源：

- symbolNode 【从symbol复制而来】
- overrides中的数据 【主动更改保存的数据，会保存自己和子级的数据】

The screenshot shows the Figma interface with two instances of a symbol. The left instance is selected, highlighted with a red border. The right panel displays the state data for the selected instance. The '当前选中节点detachState数据' (Current Selected Node detachState Data) panel shows the following JSON structure:

```
root": { "w": 355, "h": 419, "id": "0:12", "isLocked": false, "isVisible": true, "opacity": 1, "derivedSymbolData": { "10:12;0:8": { "x": 10, "y": 10, "w": 35, "h": 35, "id": "0:11" } }, "symbolOverrides": { "10:12;0:8": { "x": 10, "y": 10, "w": 35, "h": 35, "id": "0:11" } }, "uniformScaleFactor": 1, "symbolId": "0:11", "type": "instance", "name": "实例 2", "transform": { "x": 45, "y": 150, "w": 355, "h": 355, "angle": 0 }, "parentIndex": 0, "clipContent": true }
```

The '当前选中节点几何位置信息' (Current Selected Node Geometric Position Information) panel shows the following JSON structure:

```
"root": { "x": 45, "y": 150, "w": 355, "h": 355, "angle": 0 }
```

The Figma interface also shows a timeline at the bottom with frame 1 and frame 2.

The screenshot shows the Figma interface with two instances of a symbol. The left instance is selected, highlighted with a red border. The right panel displays the state data for the selected instance. The '当前选中节点detachState数据' (Current Selected Node detachState Data) panel shows the following JSON structure:

```
root": { "w": 377, "h": 247, "id": "0:8", "isLocked": false, "isVisible": true, "opacity": 1, "derivedSymbolData": { "0:8;0:3": { "x": 10, "y": 10, "w": 35, "h": 35, "id": "0:6" } }, "symbolOverrides": { "0:8;0:3": { "x": 10, "y": 10, "w": 35, "h": 35, "id": "0:6" } }, "uniformScaleFactor": 1, "symbolId": "0:6", "type": "instance", "name": "实例 1", "transform": { "x": 45, "y": 150, "w": 377, "h": 247, "angle": 0 }, "parentIndex": 0, "clipContent": true }
```

The '当前选中节点几何位置信息' (Current Selected Node Geometric Position Information) panel shows the following JSON structure:

```
"root": { "x": 45, "y": 150, "w": 377, "h": 247, "angle": 0 }
```

The Figma interface also shows a timeline at the bottom with frame 1 and frame 2.

Symbol(cache_rotation): 0

- ▶ **Symbol(cache_rt): Float32Array(9) [1, 0, 0, 0, 0, 1, 0,**
- ▶ **Symbol(cache_strokeGeometry): [{}]**
- Symbol(cache_symbolMasterId): "0:11"**
- absoluteAABB: (...)**
- absolutePosition: (...)**
- apivot: (...)**
- apoints: (...)**
- at: (...)**
- baseId: undefined**
- baseNode: (...)**
- blendMode: (...)**
- blurEffects: (...)**
- borderBottomWeight: (...)**
- borderLeftWeight: (...)**
- borderRightWeight: (...)**

2.instanceChild非instance类型节点【不会产生替换】

正常情况: baseNode, 最外层overdata

3.instanceChild且是instance节点

instanceChild直接拿state, 唯一错的就是这个id不对, 这里是baseNode的overrides, 还有最外层instance上的

当前选中节点detachState数据

```

root: { 15 items
  "w": 174.1848425865173
  "h": 247
  "id": "10:12:0:8"
  "isLocked": false
  "isVisible": true
  "opacity": 1
  "uniformScaleFactor": 1
  "symbolId": "0:16"
  "type": "instance"
  "name": "实例 1"
  > "transform": {...} 6 items
  > "parentIndex": {...} 2 items
  "clipContent": true
  > "derivedSymbolData": { 1 item
    "symbolOverrides": { 1 item
      "10:12:0:8": { 5 items
        > "fillPaints": {...} 1 item
        > "strokePaints": {...} 1 item
        "strokeWeight": 10
        "w": 574.1848425865173
        "h": 247
      }
    }
  }
}

```

当前选中节点几何位置信息

```

root: { 6 items
  "x": 45
  "y": 68
  "w": 574.1848425865173
  "h": 247
  "rotation": 0
  > "transform": {...} 6 items

```

当前选中节点detachState数据

```

root: { 15 items
  "w": 855
  "h": 869
  "id": "0:16"
  "isLocked": false
  "isVisible": true
  "opacity": 1
  > "derivedSymbolData": { 0 items
  "symbolOverrides": { 4 items
    "0:16": {...} 3 items
    "10:16:0:12": {...} 2 items
    > "10:16:0:12:10:12:0:8": { 2 items
      "10:16:0:12:10:12:0:8:10:0:8:3": {...} 2 items
    }
  }
}

```

当前选中节点几何位置信息

```

root: { 6 items
  "x": 577
  "y": 1723
  "w": 855
  "h": 869
  "rotation": 0
  > "transform": {...} 6 items

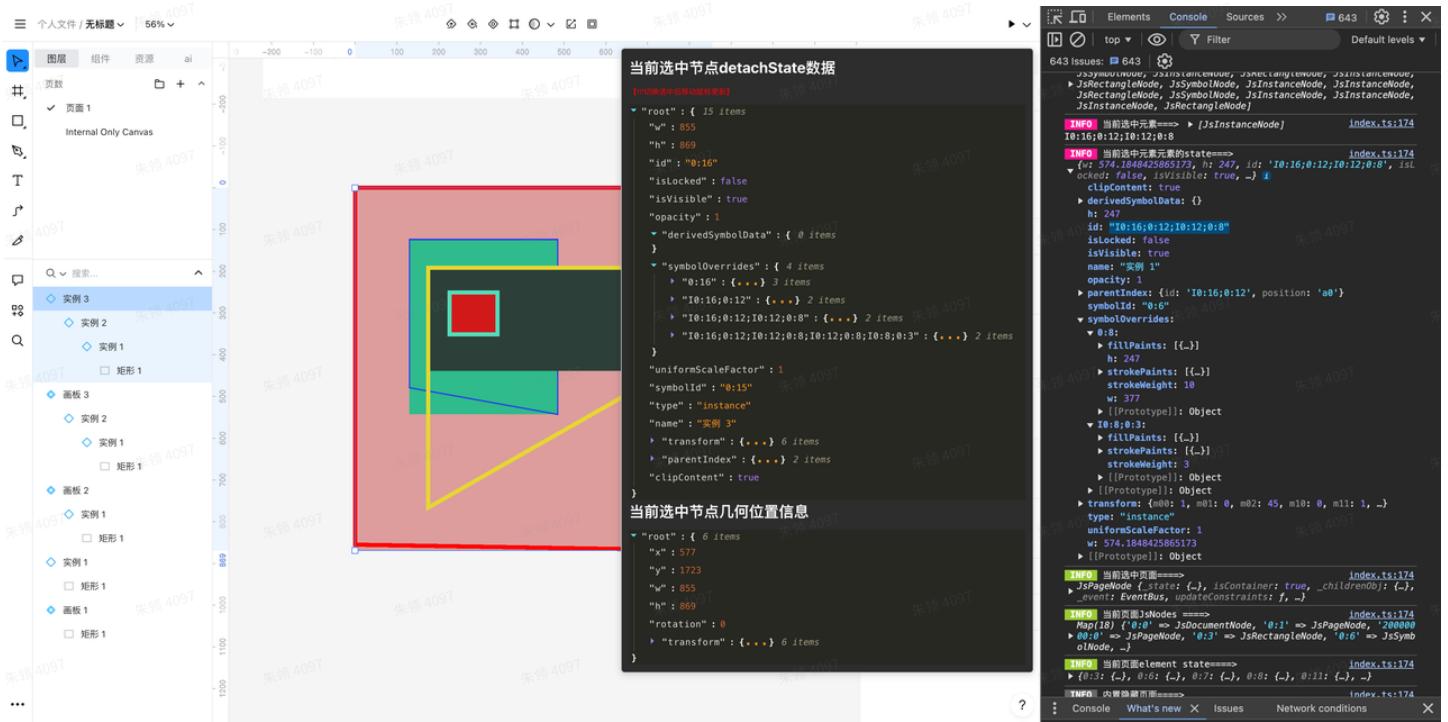
```

周一一看这里

getState中已经处理过，取值拿取顺序，只有overrides是不对的，搞对就行

- instanceChild类型的instance的数据直接拿getState然后将id替换为新的id
- 最外层overrides上的数据拿key中包含当前节点id的数据，然后将前缀id替换

- 将所有数据塞入新的overrides中即可。



问题记录

这里为什么发生替换的instance要改baseId?

```

1 if (isInstanceChildNode(node)) {
2     let baseId = node.baseId!;
3     if (isInstanceNode(node) && node.overrideSymbolId) {
4         baseId = node.overrideSymbolId;
5     }
6     baseNode = this.getNodeById(baseId) as IJsContainerNode;
7 };
8

```