【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

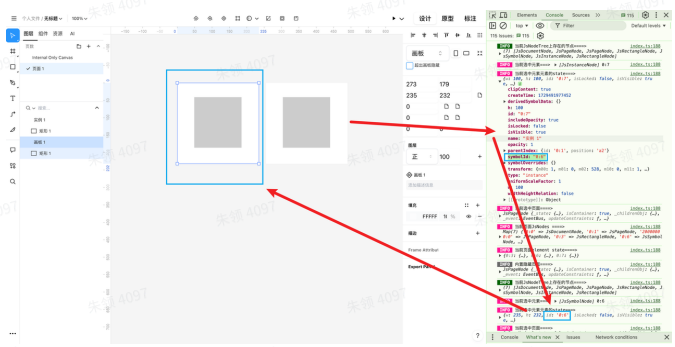
- 创建
- 删除
- 恢复
- 推送
- 重置
- 分离
- 替换

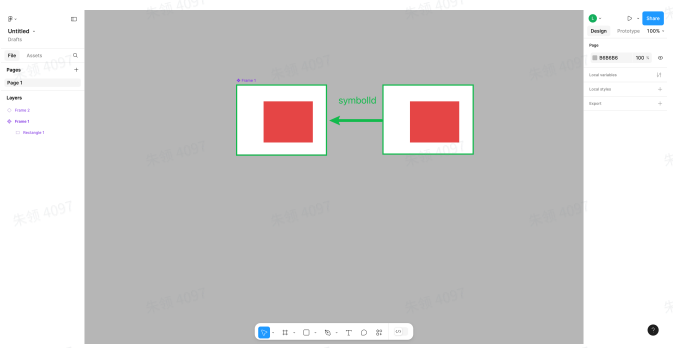
关联关系整理 ✅

当前节点id信息

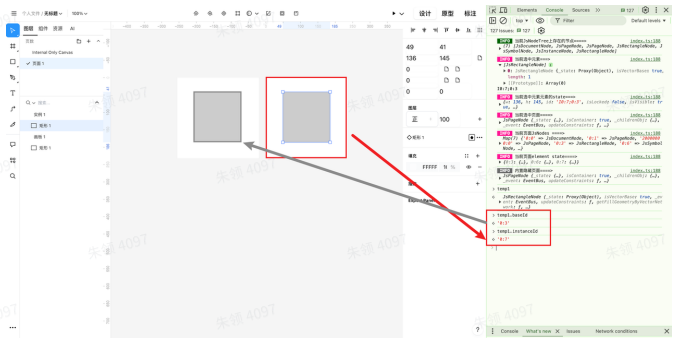

```
"root" : { 9 items
  "id" : "0:12"
  "parentId" : "0:1"
  "pageId" : "0:1"
  "instanceId" : undefined
  "baseId" : undefined
  "symbolId" : undefined
  "symbolMasterId" : ""
  "overrideSymbolId" : undefined
  "symbolRelationSet" : [ 2 items
    0 : "0:13"
    1 : "0:14"
  ]
}
```

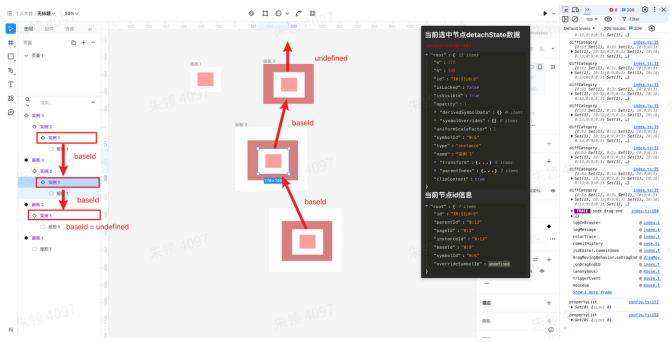

- symbolId：instance指向symbol节点的关联，这个是会存到state中的



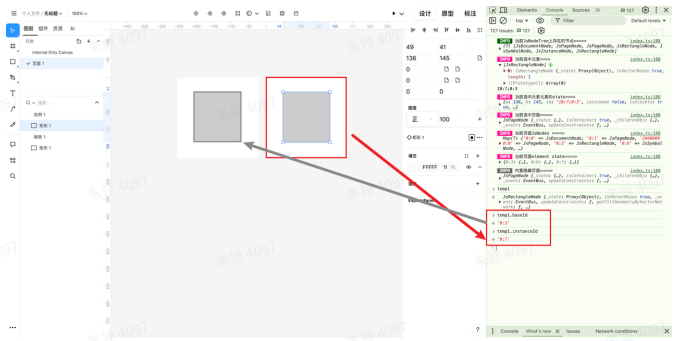


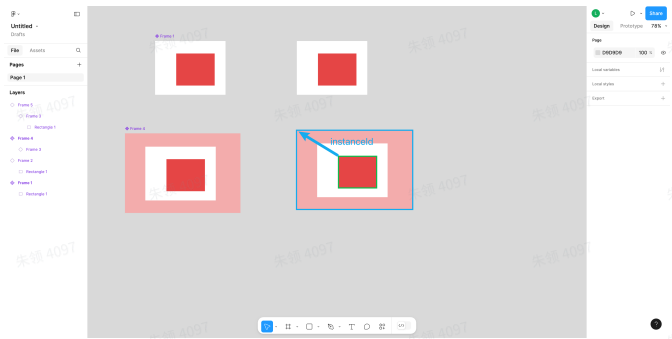
- baseId: symbol子集的id，这个是临时jsNode上的数据，标识symbol子集的关联关系



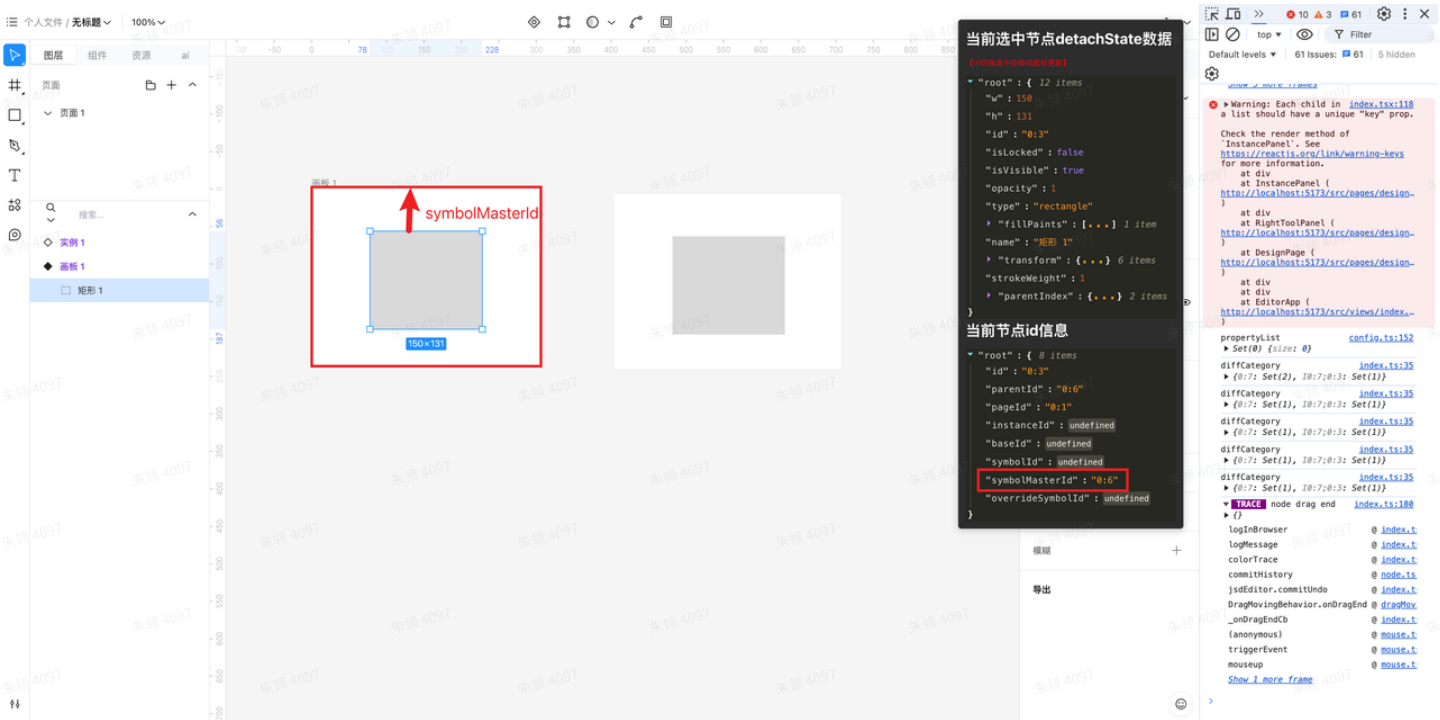


- instanceId: 判断是否是instanceIdchild，instanceIdchild跟最外层instance的关系，这个是临时jsNode上的数据

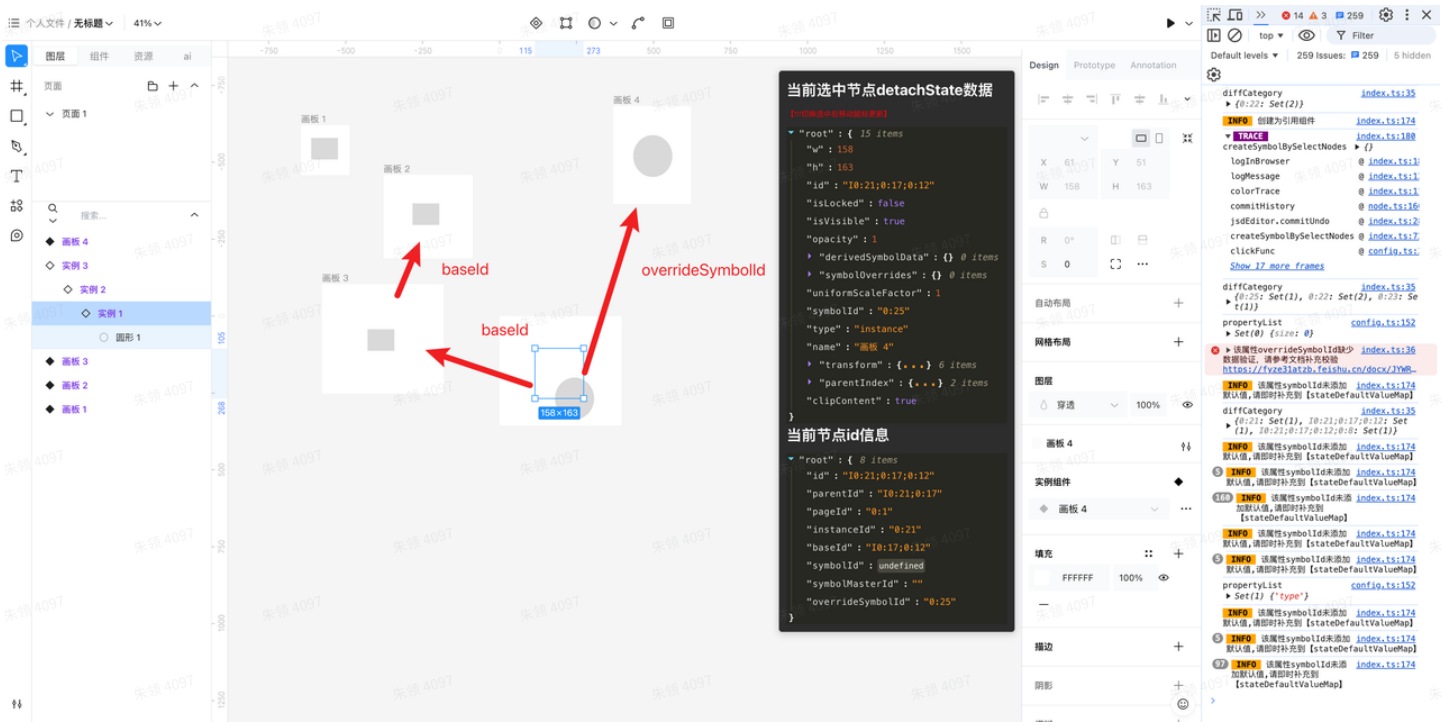




- symbolMasterId: symbolChild & symbol的关系用symbolMasterId，这个是临时jsNode上的数据



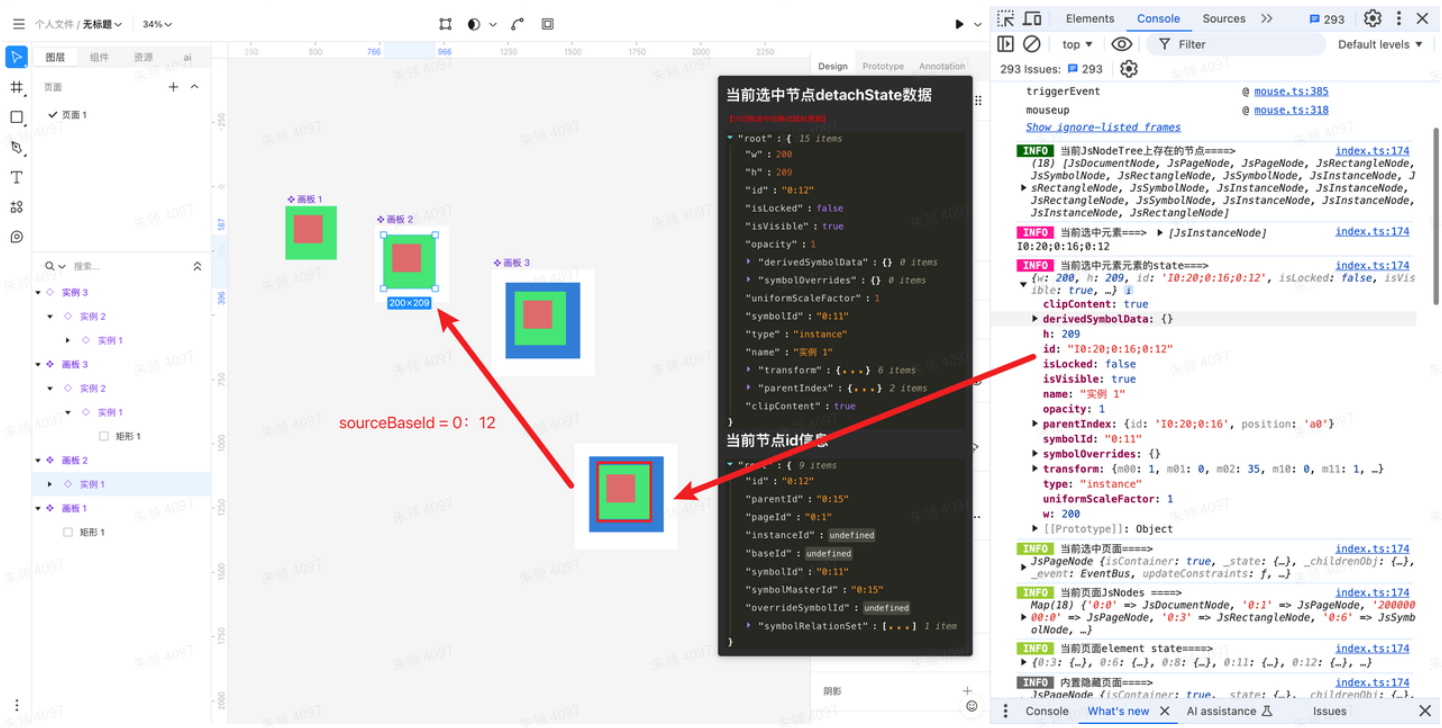
- overrsSymbolId：替换组件用overrsSymbolId，用于指向替换后的symbol，替换child类型instance时overrsSymbolId是放置在override中的，替换最外层instance则直接替换symbolId



- symbolRelationSet：与symbol关联的instance的id列表

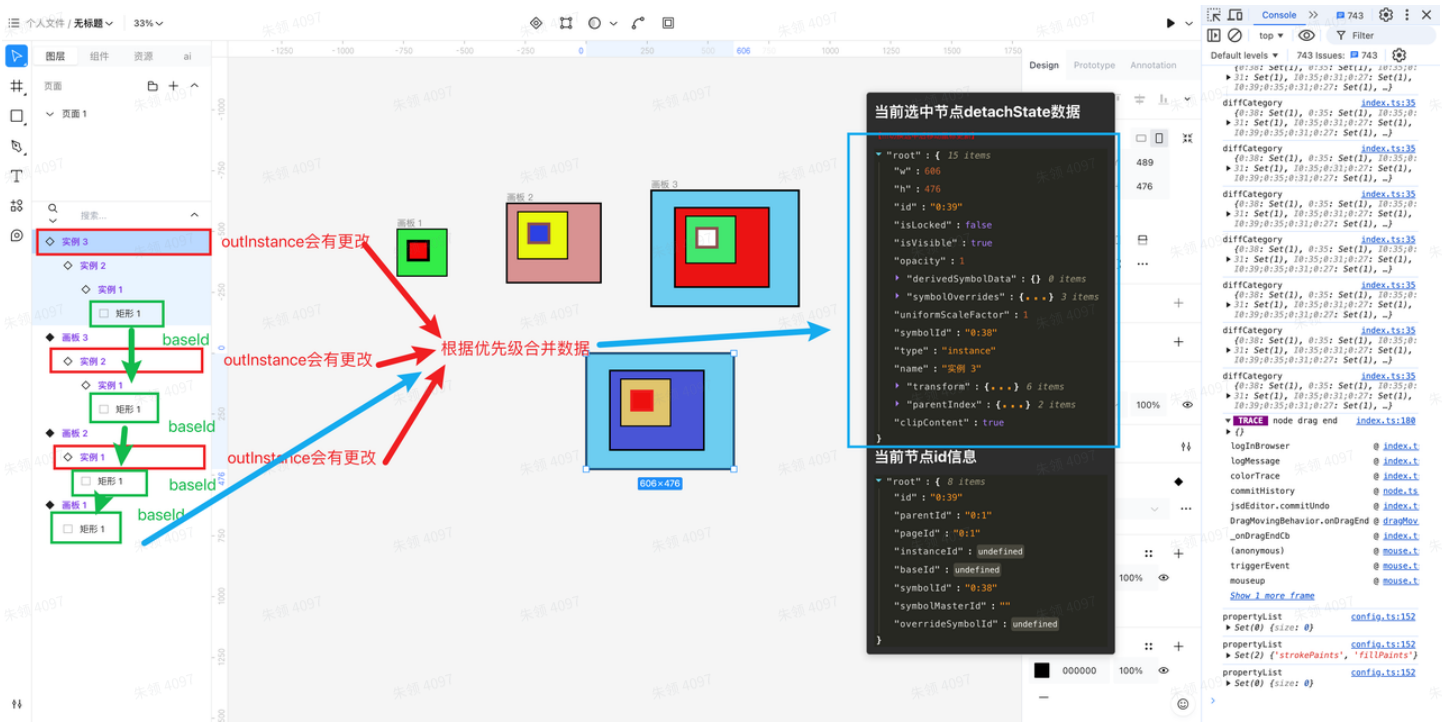
几个核心node

sourceBaseNode：最原始的baseId对应的节点



instanceChild数据来源优先级 ✅

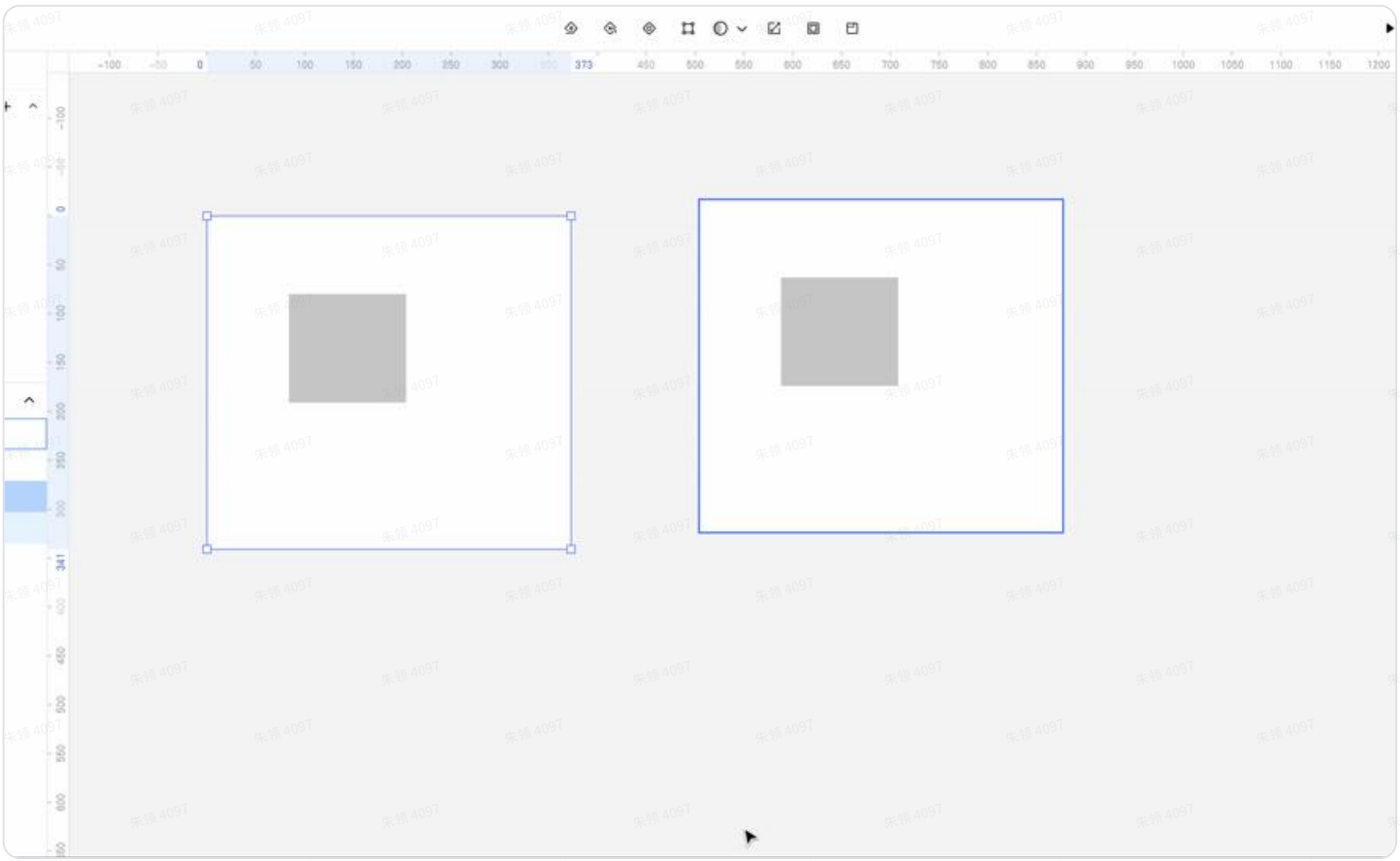
1 最外层instance上的overrides上的数据 > baseNode上的overrides数据 > symbol上的数据



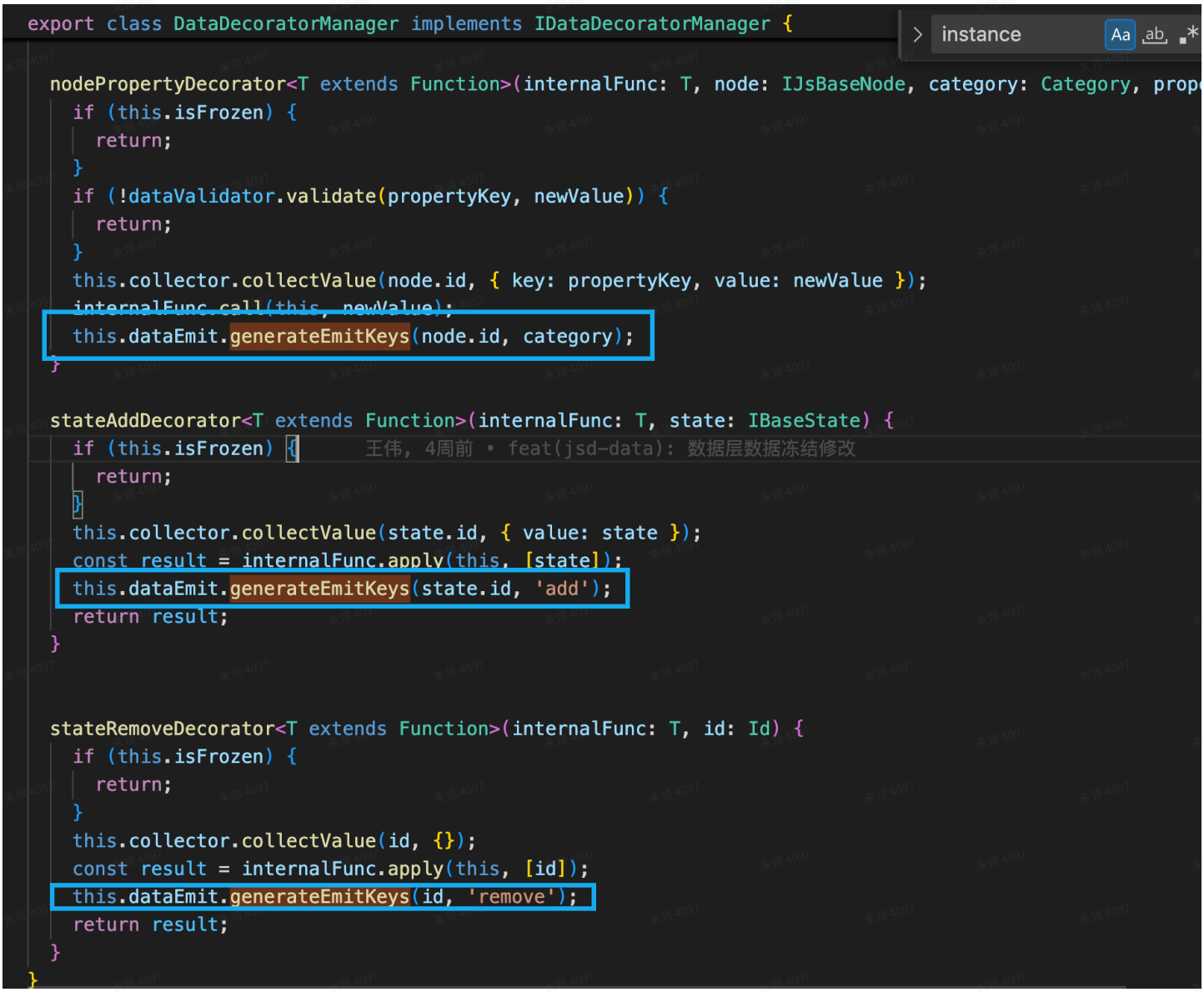
联动效果实现原理:SymbolRelationManager ✅

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



数据存储：对于instance的代理generateInstanceState

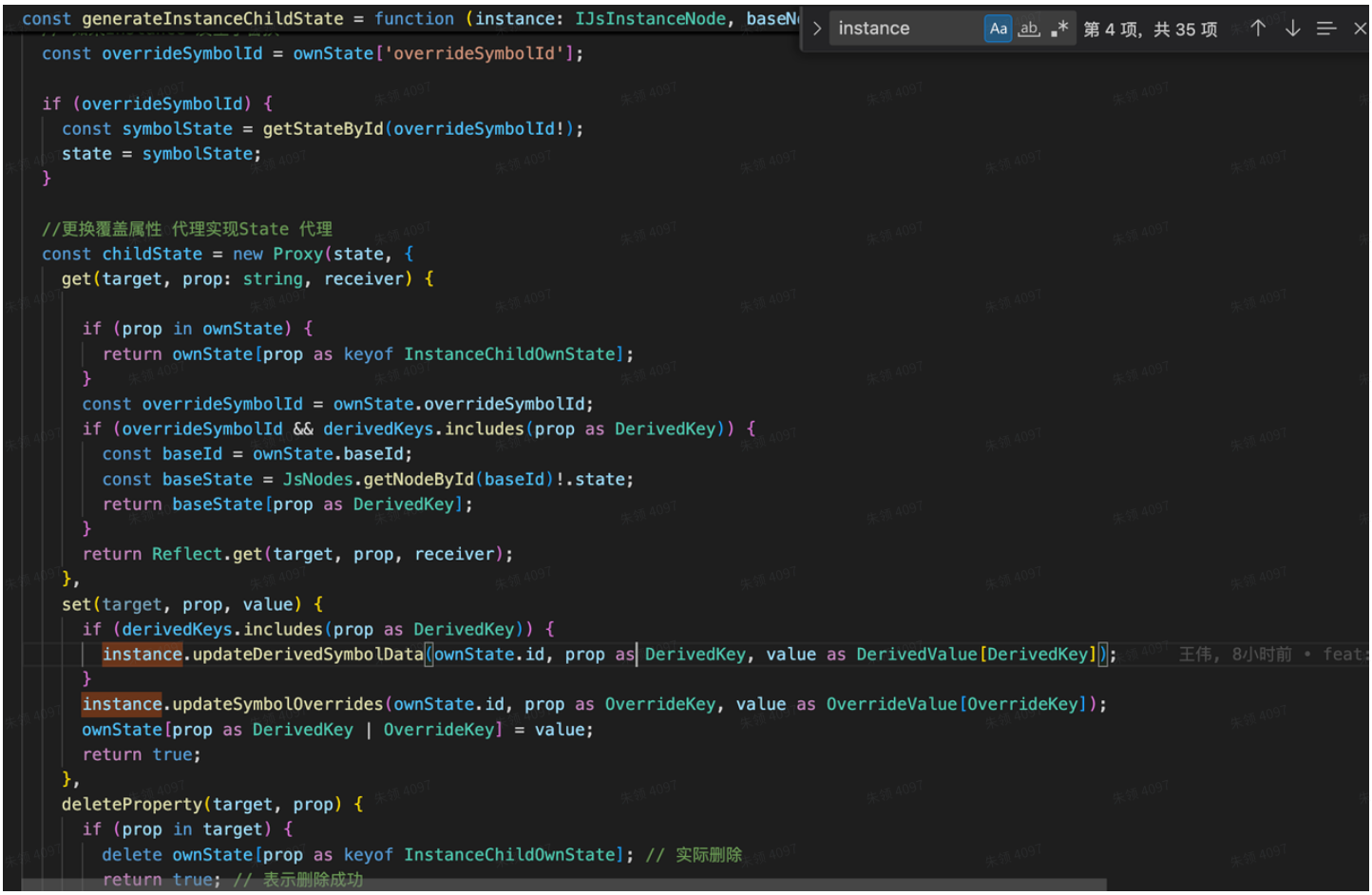
数据一部分存储在本身的overrides中更新&获取同时代理

```
const generateInstanceState = function (instanceState: IInstanceState) {

  //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
      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;
    }
  });
```

getOverrideValue Aa ab .*

数据存储：对于instanceChild的代理generateInstanceChildState

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



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

```
// 递归处理子集
(function deepHandle(baseNode: IJsContainerNode, parent: IJsContainerNode)
{
  baseNode.children.forEach((child) => {
    const id = generateInstanceChildId(parent, child);
    // 处理替换逻辑
    if (isInstanceNode(child)) {
      const overrideSymbolId = instance.getOverrideValue(id)
['overrideSymbolId'];
      const overrideSymbolNode = self.getNodeById(overrideSymbolId!) as
IJsSymbolNode;
      if (overrideSymbolNode) {
        const instanceChild = self.constructInstanceChild(instance, child,
parent);
        if (isContainerNode(child)) {
          deepHandle(overrideSymbolNode, instanceChild as
IJsContainerNode);
        }
        return true;
      }
```

```
        }
        const instanceChild = self.constructInstanceChild(instance, child,
parent);
        // 递归创建其他节点
        if (isContainerNode(child)) {
          deepHandle(child, instanceChild as IJsContainerNode);
        }
      });
    }(symbol, nearestInstance));
```

```
static constructInstanceChild(instance: IJsInstanceNode, baseNode:
IJsBaseNode, parent: IJsContainerNode) {
    const childState = generateInstanceChildState(instance, baseNode, parent);
    // 构造节点
    const childNode = this.constructNode(childState, true) as IJsBaseNode;
    if (childNode) {
      // 重新构建子父级关系
      parent.appendChildNode(childNode);
    }
    return childNode;
  }
```

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：generateInstanceChildState

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);
}
```

```ts
// 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;      王伟, 3周前

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

  return state;
};
```

generateInstanceChildState

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

```ts
const generateInstanceChildState = function (instance: IJsInstanceNode, baseNode: IJsBaseNode, parent: IJsContainerNode) {
  // 代理baseNode的state和outInstance上overrides中的数据
  let state = baseNode.state as IBaseState;
  const id = generateInstanceChildId(parent, baseNode);
```

```
  // 判断是不是 Instance  并且是否存在过替换组件
  // 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
  const overrideValue = instance.getOverrideValue(id);
  const derivedValue = instance.getDerivedValue(id);

  // 这样将overrides聚合是因为避免多层嵌套Proxy导致性能问题
  const ownState: InstanceChildOwnState = {
    baseId: state.id,
    id: id,
    instanceId: instance.id,
    parentIndex: {
      id: parent.id,
      position: state.parentIndex.position
    },
    ...derivedValue,
    ...overrideValue,
  };

  // 这是一个业务要求，baseNode的锁定和显示隐藏要通信
  if (state.type === 'instance') {
    ownState.isLocked = state.isLocked;
    ownState.isVisible = state.isVisible;
  }

  // 如果Instance 发生了替换
  const overrideSymbolId = ownState['overrideSymbolId'];

  // 发生替换后只有w,h,transfrom走baseNode，其他的走替换后的symbol
  if (overrideSymbolId) {
    if ((state as IInstanceState).symbolId) {
      ownState.symbolId = (state as IInstanceState).symbolId;
    }
    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) {
        if (prop === 'parentIndex') {
          const parentIndex = ownState[prop as keyof InstanceChildOwnState] as
ParentIndex;
          parentIndex.position = state.parentIndex.position;
          return parentIndex;
        }
```

```
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,
    value as DerivedValue[DerivedKey]);
65        }
66        instance.updateSymbolOverrides(ownState.id, prop as OverrideKey, value as
     OverrideValue[OverrideKey]);
67        ownState[prop as DerivedKey | OverrideKey] = value;
68        return true;
69      },
70      deleteProperty(target, prop) {
71        if (prop in ownState) {
72          delete ownState[prop as keyof InstanceChildOwnState]; // 实际删除
73          return true; // 表示删除成功
74        } else {
75          return true; // 表示删除失败
76        }
77      }
78    });
79
80    return childState;
81  };
```

数据通信

symbol改动是如何通信instance的

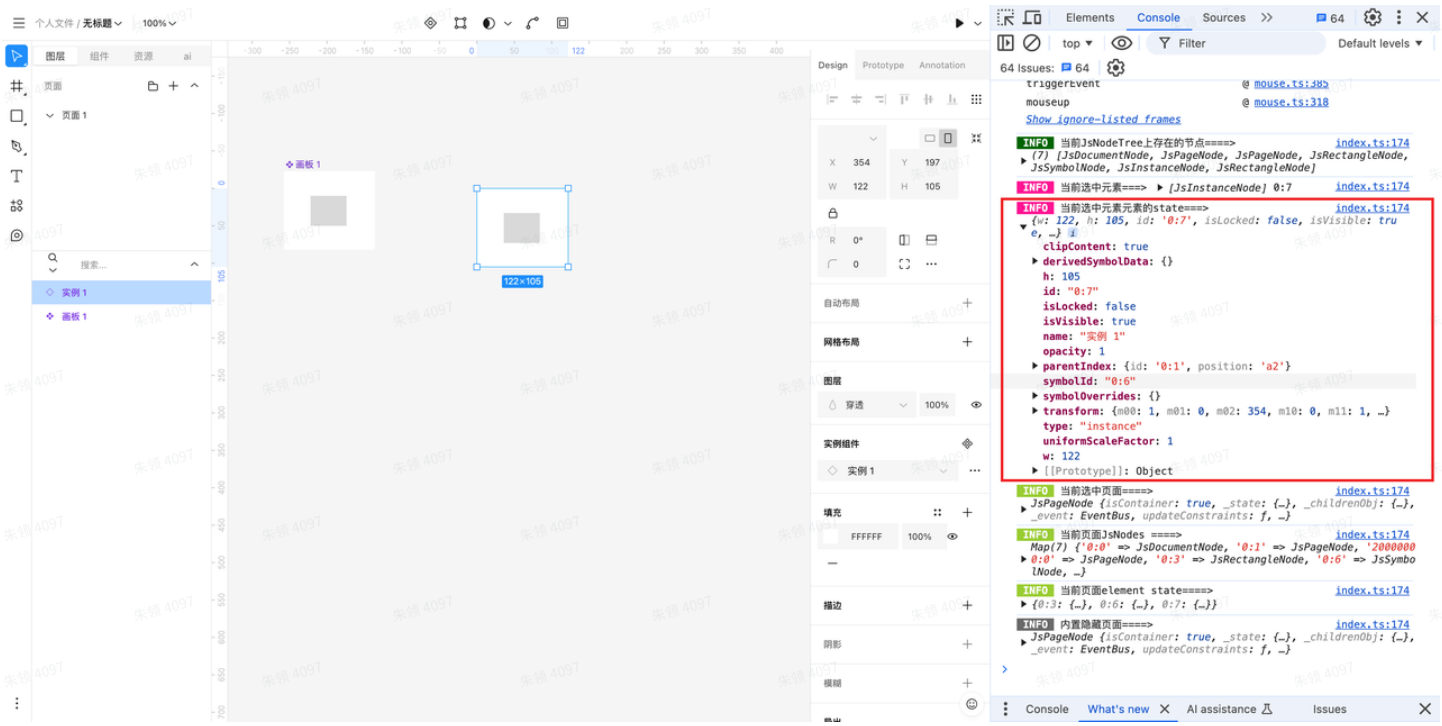
主要是在构建instance节点时，代理instance其本身的state数据，instance是具备实体state数据的getter拦截：

- 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 ✅

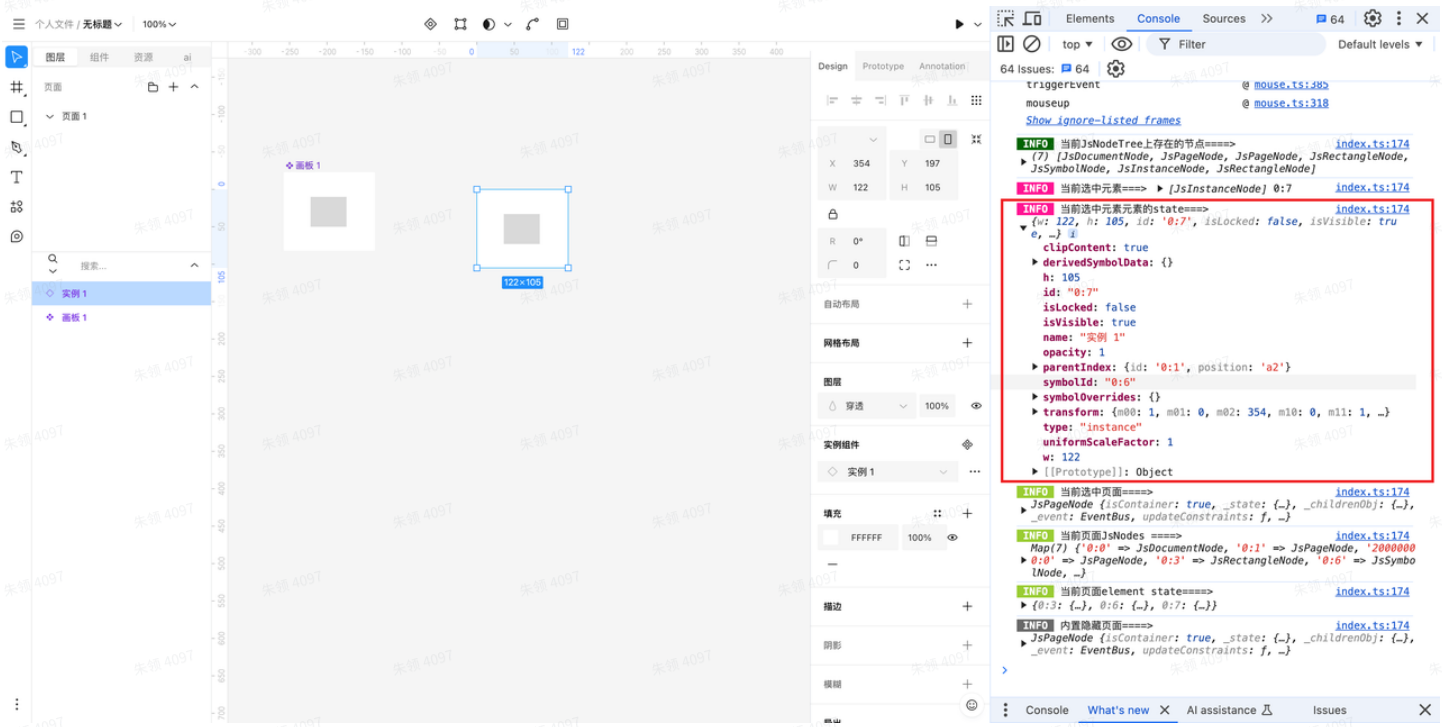
创建symbol

- 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

```ts
/**
 * 创建引用组件
 * @param id
 * @returns
 */
export const createSymbolNode = (frameNode: IJsFrameNode): IJsSymbolNode | undefined => {
  if (!frameNode) {
    return;
  }
  if (frameNode.type !== 'frame') {
    logHelper.error('传入节点存在非frame，仅frame可创建为symbol');
    return;
  }
  const basicSymbol = getBasicSymbol();
  const oldStateData = frameNode.getState();
  // 构建新结构
  const newSymbolData = {
    ...basicSymbol,
    ...oldStateData,
    id: generateIncID(EIncIdType.Element),
    type: 'symbol'
  } as ISymbolState;
  // 增加新节点
  if (oldStateData) {
    JsNodes.addNode(newSymbolData);
  }
  // 构造新节点
  const symbolNode = JsNodes.getNodeById(newSymbolData.id) as IJsSymbolNode;
  frameNode.children.forEach((node) => {
    if (!node) {
      return;
    }
    node.changeParentIndex(symbolNode.id);
  });
  // 移除旧节点
  frameNode.removeNode();
  return symbolNode;
};
```

创建instance

核心数据，用于承载变更，和关联关系




```
  derivedSymbolData: {},
  symbolOverrides: {},
  symbolId: symbolId,

  /**
  * 基础矩形默认schema数据
  * @returns {IFrameState}
  */
export function getBasicInstance(symbolId: Id): IInstanceState {
   const symbolNode = JsNodes.getNodeById(symbolId)!;
   if (!symbolNode) {
     logHelper.warn('symbolNode节点不存在');
   }
   return {
     w: symbolNode.w,
     h: symbolNode.h,
     ...getDefaultBaseSchema(),
     derivedSymbolData: {},
     symbolOverrides: {},
     uniformScaleFactor: 1,
     symbolId: symbolId,
     type: 'instance',
     name: generateDefaultLayerName('instance'),
```

```
    transform: defaultTransform({ x: symbolNode.x, y: symbolNode.y }),
    parentIndex: generateParentIndex(),
    clipContent: true,
  };
}
```

```
export function createInstance(symbolId: Id) {
  const instanceState = getBasicInstance(symbolId);
  JsNodes.addNode(instanceState);
  const symbol = JsNodes.getNodeById(symbolId)!;
  const instance = JsNodes.getNodeById(instanceState.id)!;
  instance.changePos({ x: symbol.x + symbol.w + 20, y: symbol.y });
  return instance;
}
```

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 {JsInstanceNode} 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 = generateInstanceChildId(parent, child);
        const overrideSymbolId = instance.getChildOverrideVal(id, 'symbolId');
        overrideSymbolNode = self.getNodeById(overrideSymbolId!) as IJsSymbolNode;
      }
      const instanceChild = self.constructInstanceChild(instance, child, parent, overrideSymbolNode);
      if (isContainerNode(child!) && instanceChild) {
        deepHandle(overrideSymbolNode as IJsContainerNode ?? child, instanceChild as IJsContainerNode);
      }
    });
  }(symbol, instance));
};
```

```
/**
```

```
  * 根据Id构造指定节点
  * @param {string} id - 节点ID
  * @returns {Node} - 返回构建的节点
  */
  static constructNode(elementState: IBaseState, force: boolean = false) {
    const elementId = elementState.id;
    let thisJsNode = this.nodes.get(elementId);
    if (!thisJsNode || force) {
      thisJsNode = State2JsNodeService.transform(elementState);
    }

    if (thisJsNode) {
      this.nodes.set(elementId, thisJsNode);
      const parent = thisJsNode.parent as IJsContainerNode;
      if (parent) {
        parent.addChildNode(thisJsNode);
      }
      // 构造！！！
      if (isInstanceNode(thisJsNode) && !isInstanceChildNode(thisJsNode)) {
        this.constructInstanceChildren(thisJsNode);
      }
      // 节点创建时调用触发
      thisJsNode.onCreate?.();
    }
    return thisJsNode;
  }
```

instanceChild的state数据来源于

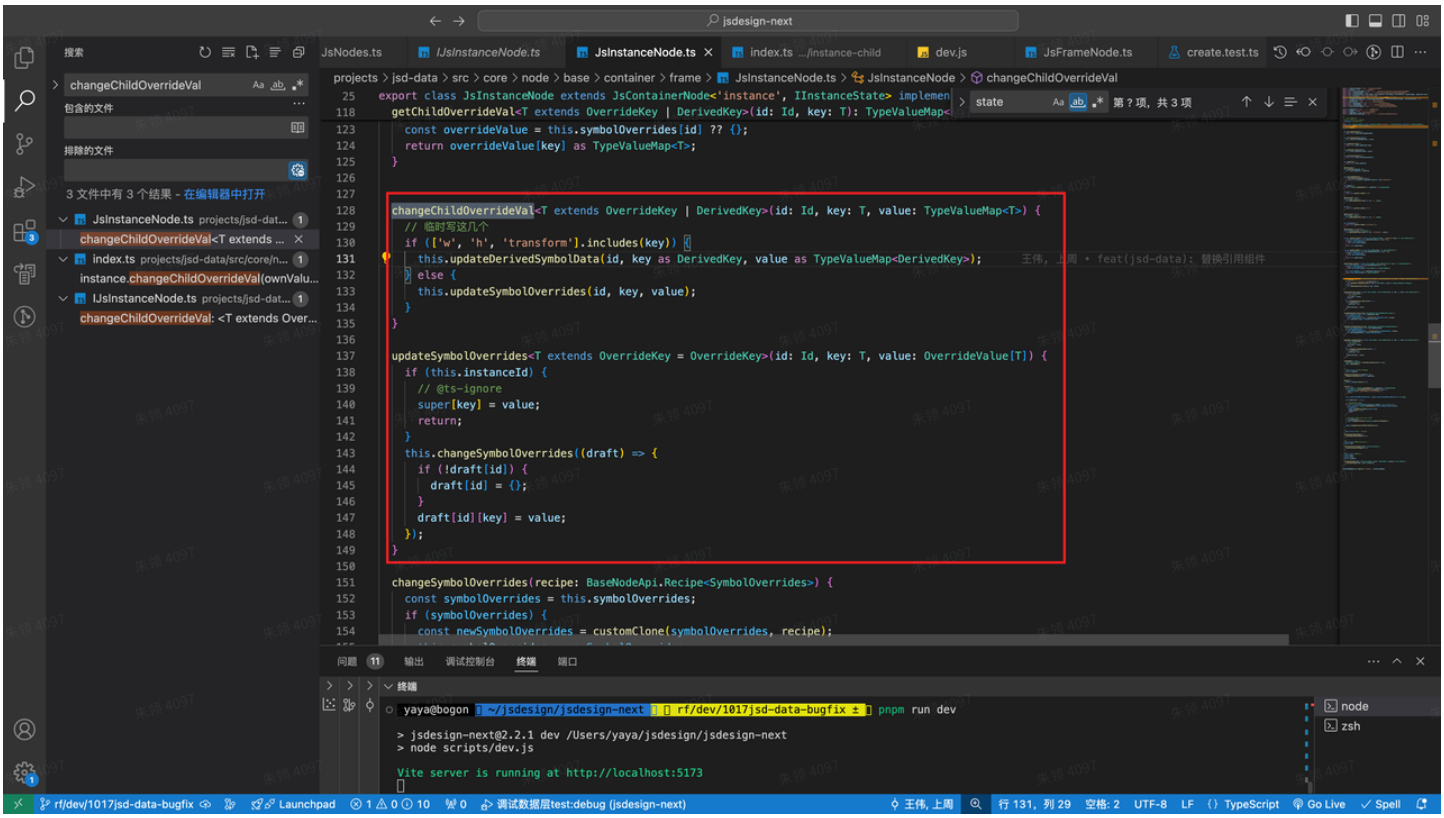
- instanceChild的state数据来源于，优先级从上往下，数据层的实现是通过代理baseNode的state去获取instanceChild的state，构造临时的jsNodes
  - 自身的映射关&父子级关系ownValue
  - 自身更改过的属性overrideVal
  - `'transform', 'w', 'h', 'type'` 从baseNode【baseId指向】上拿
  - 如果存在替换，样式类属性从替换的symbol节点上拿symbolData
  - 所有其他除以上情况外的数据均从baseNode上拿

```
export const generateInstanceChildState = function (instance: IJsInstanceNode, baseNode: IJsBaseNode, parent: IJsContainerNode, symbolNode?: IJsSymbolNode) {
  let state = baseNode.getState() as IBaseState;
  const id = generateInstanceChildId(parent, baseNode);
  let symbolData = symbolNode?.getState() ?? {};
```

```
  // 判断是不是 Instance  并且是否存在过替换组件
  // 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

  const ownValue = {
    baseId: state.id,
    id: id,
    instanceId: instance.id,
    parentIndex: {
      id: parent.id,
      position: state.parentIndex.position
    }
  };
  //更换覆盖属性 代理实现State 代理
  const childState = new Proxy(state, {
    get(target, prop: string, receiver) {
      // 优先从 overrideObj 中查找属性
      const overrideVal = instance.getChildOverrideVal(ownValue.id, prop as OverrideKey | DerivedKey);
      if (prop in ownValue) {
        // @ts-ignore
        return ownValue[prop];
      }
      if (overrideVal) {
        return overrideVal;
      }
      if (['transform', 'w', 'h', 'type'].includes(prop)) {
        return Reflect.get(target, prop, receiver);
      }
      // 从替换的组件上拿属性

      if (prop in symbolData) {
        // @ts-ignore
        return symbolData[prop];
      }
      return Reflect.get(target, prop, receiver);
    },
    set(target, prop: OverrideKey | DerivedKey, value: OverrideValue[OverrideKey]) {
      // 将新值直接设置到 overrideObj 中
      instance.changeChildOverrideVal(ownValue.id, prop, value);
      return true;
    }
  });

  return childState;
};
```

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



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

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

```
  const variant = createContainer(symbolIds, 'variant') as IJsVariantNode;

  variant.changeStateGroupPropertyValueOrders((stateGroupPropertyValueOrders) => {
    const newProperty = {
      property: 'Property 1',
      values: variant.children.map(item => item.name)
    };
    stateGroupPropertyValueOrders.push(newProperty);
  });

  return variant;
}
```

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`

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

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: { } }
    ],
    frameMaskDisabled: true,
    symbolLinks: [
      { uri: "https://www.baidu.com" }
    ],
    editInfo: { userId: "1210200424553611378", lastEditedAt: 1730084932, createdAt: 1730084932 }
  }
],
visible: true
```


```
1
```

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

```
(function traverseChildren(outermostInstance: IJsBaseNode, parent: IJsBaseNode) {
    outermostInstance.children.forEach((child) => {
      let curNewNode;
      if (isInstanceNode(child) && detachInstanceId.includes(child?.id)) {
        curNewNode = convertInstanceToFrame(child, parent);
      } else if (isInstanceNode(child)) {
        curNewNode = child.cloneNode();
        curNewNode.changeParentIndex(parent.id);
      } else {
        const oldData = child.detachState;
        curNewNode = createNodeByState(child.type, {
          ...oldData as UnionBaseState,
          parentIndex: generateParentIndex(parent.id),
          id: generateIncID(EIncIdType.Element),
        });
      }
      if (detachInstanceId.includes(child.id) || !isInstanceNode(child)) {
        traverseChildren(child, curNewNode);
      }
    });
  }(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

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

```
    if (!JsNodeTree.checkNodeIsDelete(symbol)) {
      logHelper.error('symbol节点未在internalPage中');
      return;
    }
    symbol.changeParentIndex(oldPageParentExist ? originPageId[0] :
JsNodeTree.getCurrentPageId());
    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

```
/**
 * @description 重置某个child节点的所有覆盖属性
 * @param node
 * @param child
 * @param property
 */
export const resetChildChange = function (instance: IJsInstanceNode, child:
IBaseNode) {
  const categoryList = getResetChangePropertyById(instance, child.id);
  instance.changeSymbolOverrides((draft) => {
    if (draft[child.id]) {
      if (draft[child.id].overrideSymbolId) {
        (child as IJsInstanceNode).traverseChildren((_child) => {
          delete draft[_child.id];
          _child.removeNode();
        });
      }
      delete draft[child.id];
    }
  });
  categoryList.forEach((category) => {
    internalApi.dataEmit.generateEmitKeys(child.id, category);
  });
};
```

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
   a. 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
   b. 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

```
get overrideSymbolId() {
    return this.instance?.getOverrideValue(this.id)['overrideSymbolId']!;
  }
@DataTracker('type')
set overrideSymbolId(symbolId: Id) {
    if (this.instanceId) {
      this.state.overrideSymbolId = symbolId;
      generateSymbolRelation(this as IJsBaseNode);
    }
}
```

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

```
14
15    // 获取名字-层级对应的覆盖属性
16    const levelNameOverrideValue = saveLevelNameOverrideValue(currentInstance);
17
18    // 保留缓存属性覆盖属性
19    saveCacheOverrides(currentInstance, instance.symbolOverrides);
20
21    // 覆盖 w h 属性到新的Instance
22    if (!isInstanceChildNode(currentInstance)) {
23      const overridesValues = instance.symbolOverrides[currentInstance.id] ?? {};
24      if (!overridesValues['w']) {
25        instance.stateW = symbolNode.w;
26      }
27      if (!overridesValues['h']) {
28        instance.stateH = symbolNode.h;
29      }
30    }
31
32    // 替换为原来的symbolId 指向时 逻辑处理
33    if (currentInstance.symbolId === symbolId &&
  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);
59
```

```
60    // 同名同层级覆盖属性逻辑
61    coverLevelNameOverrideValue(currentInstance, instance,
    levelNameOverrideValue);
62
63    // 构造当前子级
64    constructInstance(currentInstance);
65    // JsNodes.constructInstanceChildren(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!;      王伟, 上周 • fix: 替换组件属性覆盖逻辑修复
    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,
   baseNode: IJsBaseNode, parent: IJsContainerNode, symbolNode?: IJsSymbolNode) {
2    let state = baseNode.getState() as IBaseState;
3    const id = generateInstanceChildId(parent, baseNode);
4    let symbolData = symbolNode?.getState() ?? {};
5
6    // 判断是不是 Instance  并且是否存在过替换组件
```

```
  // 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

  const ownValue = {
    baseId: state.id,
    id: id,
    instanceId: instance.id,
    parentIndex: {
      id: parent.id,
      position: state.parentIndex.position
    }
  };
  //更换覆盖属性 代理实现State 代理
  const childState = new Proxy(state, {
    get(target, prop: string, receiver) {
      // 优先从 overrideObj 中查找属性
      const overrideVal = instance.getChildOverrideVal(ownValue.id, prop as OverrideKey | DerivedKey);
      if (prop in ownValue) {
        // @ts-ignore
        return ownValue[prop];
      }
      if (!isUndefined(overrideVal)) {
        return overrideVal;
      }
      if (['transform', 'w', 'h', 'type'].includes(prop)) {
        return Reflect.get(target, prop, receiver);
      }
      // 从替换的组件上拿属性

      if (prop in symbolData) {
        // @ts-ignore
        return symbolData[prop];
      }
      return Reflect.get(target, prop, receiver);
    },
    set(target, prop: OverrideKey | DerivedKey, value: OverrideValue[OverrideKey]) {
      // 将新值直接设置到 overrideObj 中
      instance.changeChildOverrideVal(ownValue.id, prop, value);
      return true;
    }
  });

  return childState;
};

```

```ts
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;
  }       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 baseId，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 baseId&symbol，最外层overdata， 其他来自 replaceSymbol

其他节点【不会产生替换】

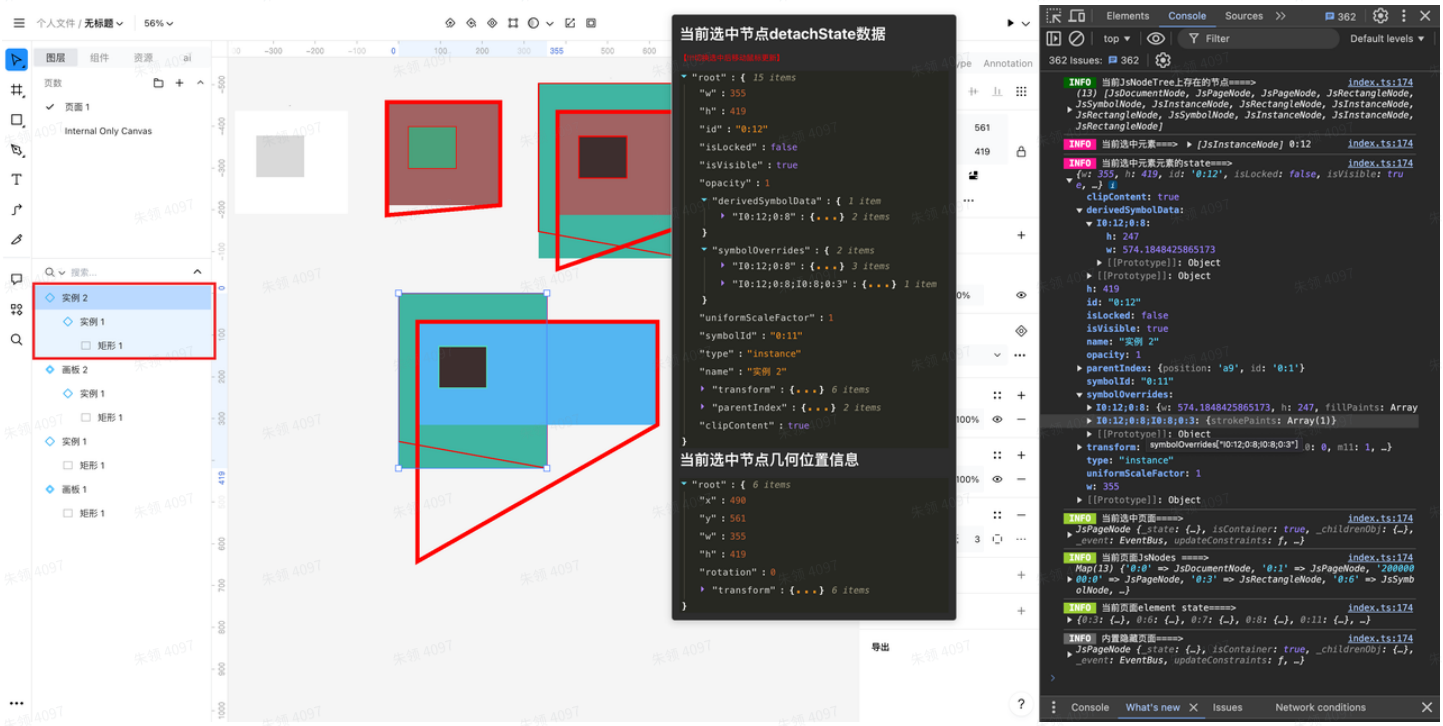
正常情况：baseNode，最外层overdata

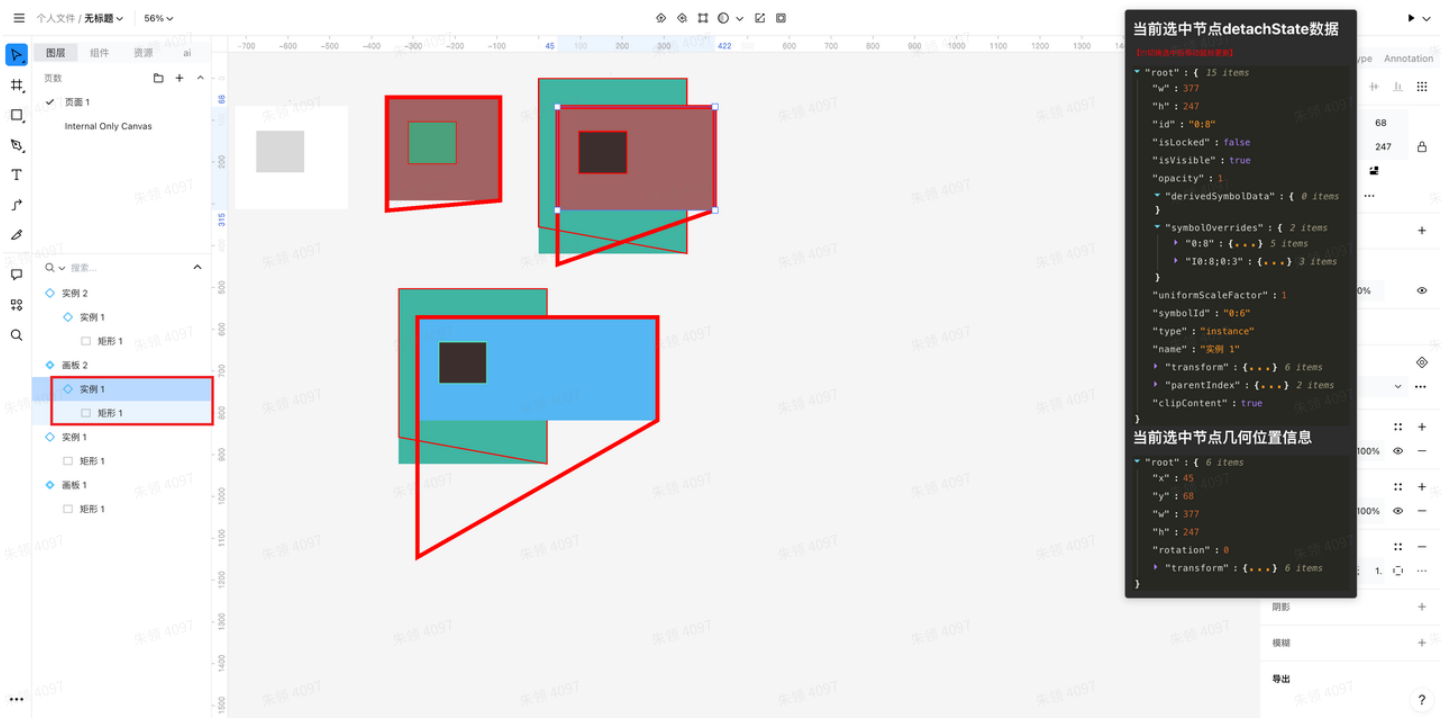
记录下情况

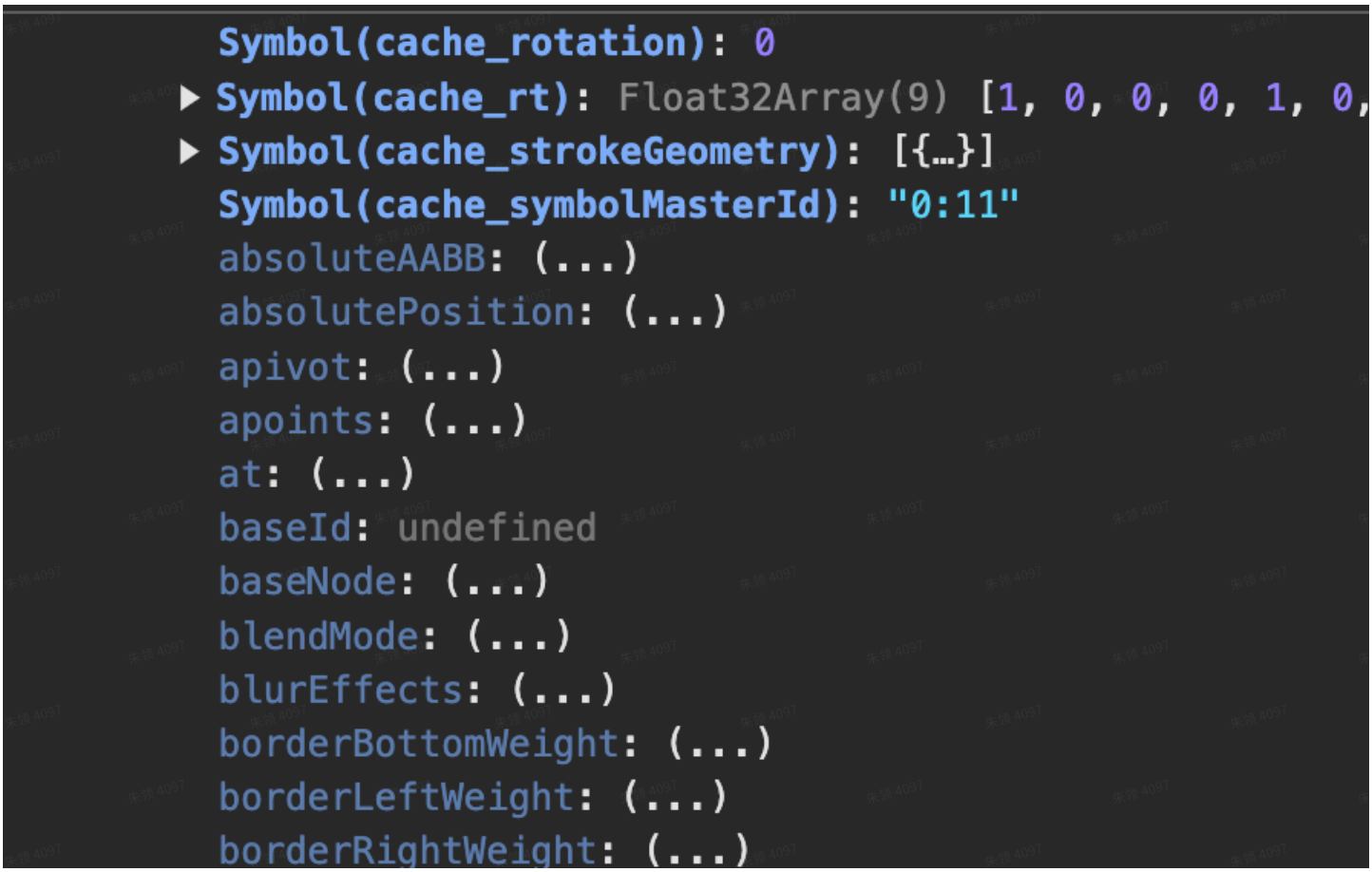
1.单独的instance，非instanceChild中的instance，无baseId

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】





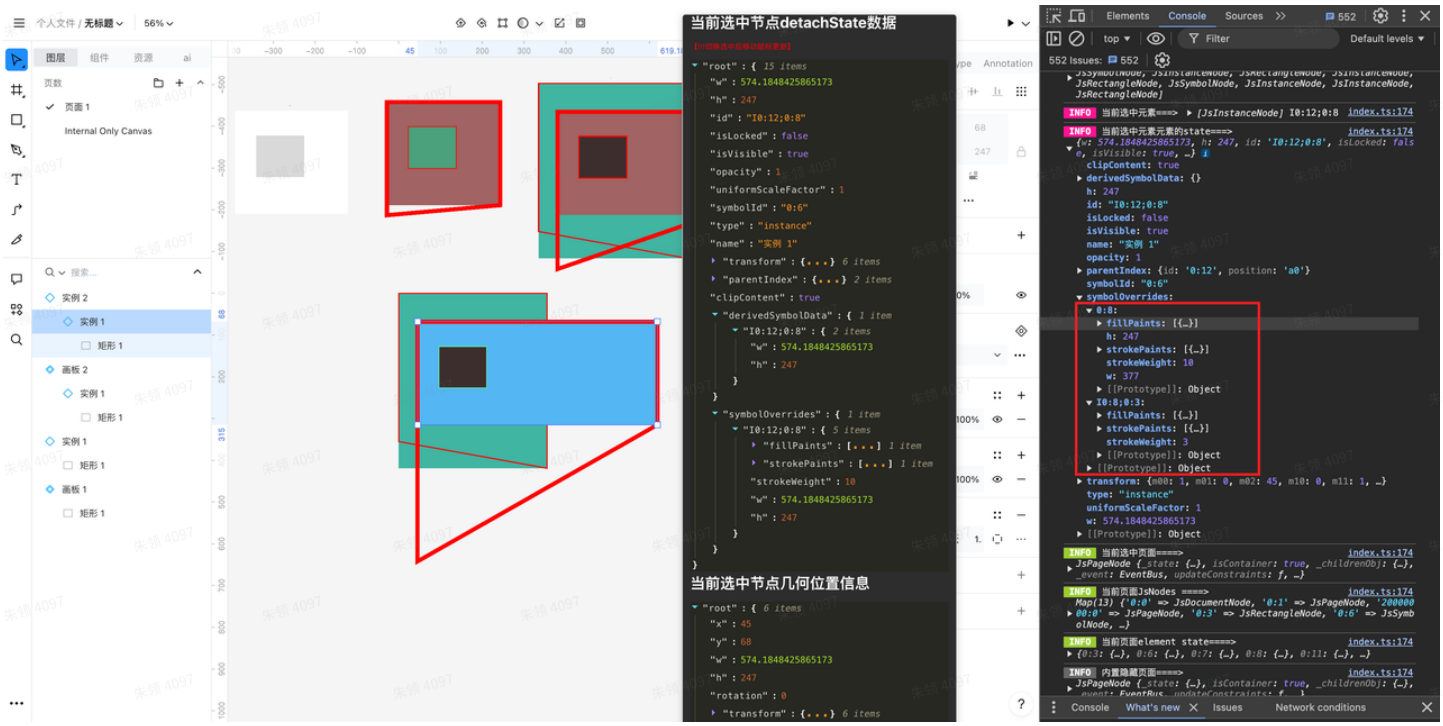


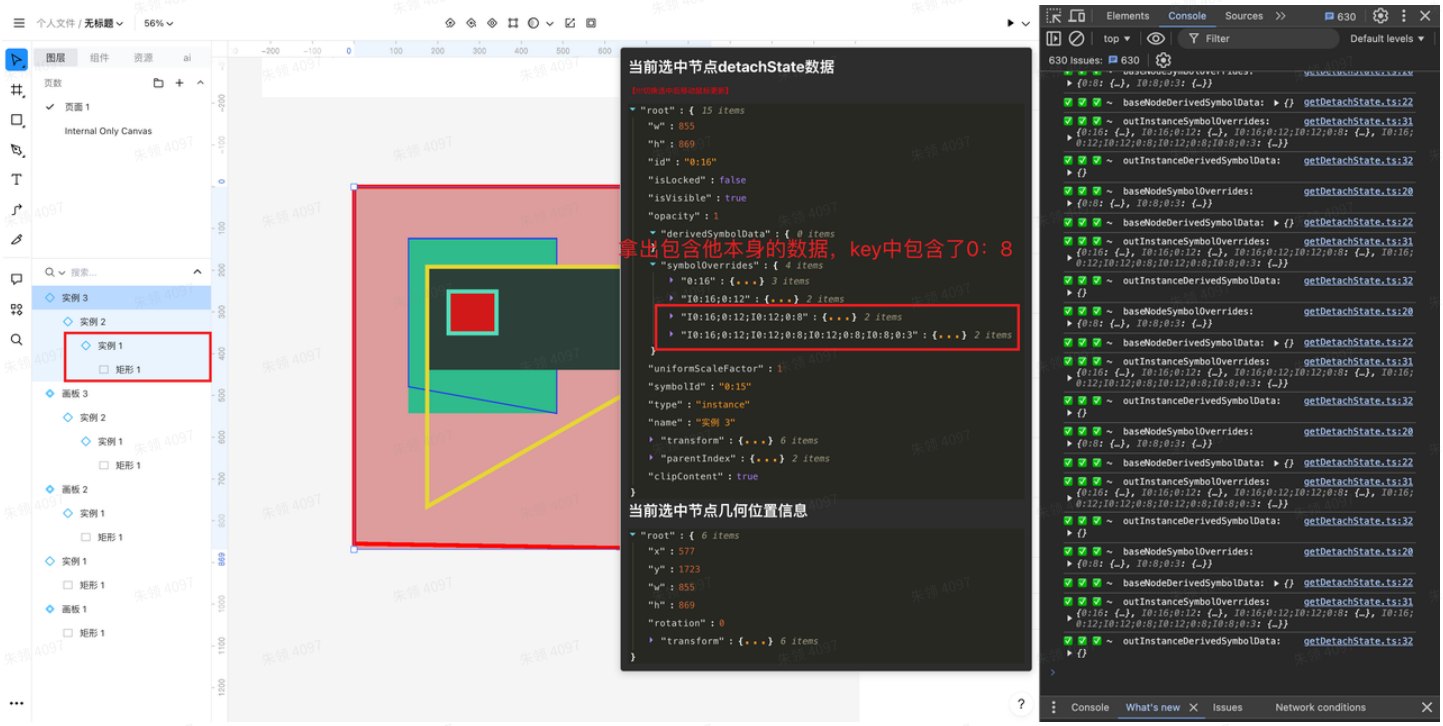
2.instanceChild非instance类型节点【不会产生替换】

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



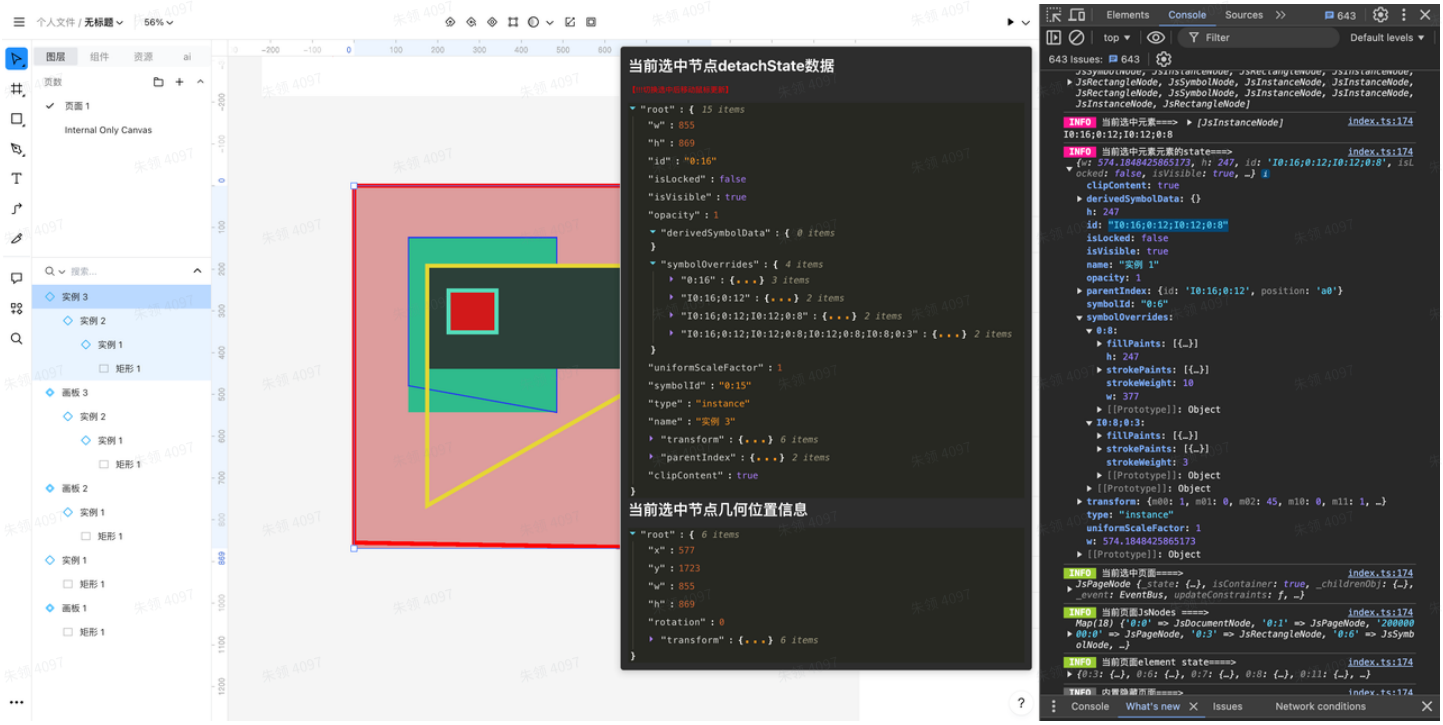


周一看这里

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?

```
if (isInstanceChildNode(node)) {
      let baseId = node.baseId!;
      if (isInstanceNode(node) && node.overrideSymbolId) {
        baseId = node.overrideSymbolId;
      }
      baseNode = this.getNodeById(baseId) as IJsContainerNode;
    };

```