js-modules.d.ts

declare module 'leaflet';  
declare module 'pouchdb';

index-BrDBppkm.js

chunk-BUSYA2B4.js

var \_\_getOwnPropNames = Object.getOwnPropertyNames;  
var \_\_commonJS = (cb, mod) => function \_\_require() {  
 return mod || (0, cb[\_\_getOwnPropNames(cb)[0]])((mod = { exports: {} }).exports, mod), mod.exports;  
};  
export {  
 \_\_commonJS  
};

chunk-USUGRH7V.js

function makeMap(str, expectsLowerCase) {  
 const set2 = new Set(str.split(","));  
 return expectsLowerCase ? (val) => set2.has(val.toLowerCase()) : (val) => set2.has(val);  
}  
var EMPTY\_OBJ = true ? Object.freeze({}) : {};  
var EMPTY\_ARR = true ? Object.freeze([]) : [];  
var NOOP = () => {  
};  
var NO = () => false;  
var isOn = (key) => key.charCodeAt(0) === 111 && key.charCodeAt(1) === 110 && // uppercase letter  
(key.charCodeAt(2) > 122 || key.charCodeAt(2) < 97);  
var isModelListener = (key) => key.startsWith("onUpdate:");  
var extend = Object.assign;  
var remove = (arr, el) => {  
 const i = arr.indexOf(el);  
 if (i > -1) {  
 arr.splice(i, 1);  
 }  
};  
var hasOwnProperty = Object.prototype.hasOwnProperty;  
var hasOwn = (val, key) => hasOwnProperty.call(val, key);  
var isArray = Array.isArray;  
var isMap = (val) => toTypeString(val) === "[object Map]";  
var isSet = (val) => toTypeString(val) === "[object Set]";  
var isDate = (val) => toTypeString(val) === "[object Date]";  
var isRegExp = (val) => toTypeString(val) === "[object RegExp]";  
var isFunction = (val) => typeof val === "function";  
var isString = (val) => typeof val === "string";  
var isSymbol = (val) => typeof val === "symbol";  
var isObject = (val) => val !== null && typeof val === "object";  
var isPromise = (val) => {  
 return (isObject(val) || isFunction(val)) && isFunction(val.then) && isFunction(val.catch);  
};  
var objectToString = Object.prototype.toString;  
var toTypeString = (value) => objectToString.call(value);  
var toRawType = (value) => {  
 return toTypeString(value).slice(8, -1);  
};  
var isPlainObject = (val) => toTypeString(val) === "[object Object]";  
  
var isReservedProp = makeMap(  
 ",key,ref,ref\_for,ref\_key,onVnodeBeforeMount,onVnodeMounted,onVnodeBeforeUpdate,onVnodeUpdated,onVnodeBeforeUnmount,onVnodeUnmounted"  
);  
var isBuiltInDirective = makeMap(  
 "bind,cloak,else-if,else,for,html,if,model,on,once,pre,show,slot,text,memo"  
);  
var cacheStringFunction = (fn) => {  
 const cache = /\* @\_\_PURE\_\_ \*/ Object.create(null);  
 return (str) => {  
 const hit = cache[str];  
 return hit || (cache[str] = fn(str));  
 };  
};  
var camelizeRE = /-(\w)/g;  
var camelize = cacheStringFunction((str) => {  
 return str.replace(camelizeRE, (\_, c) => c ? c.toUpperCase() : "");  
});  
var hyphenateRE = /\B([A-Z])/g;  
var hyphenate = cacheStringFunction(  
 (str) => str.replace(hyphenateRE, "-$1").toLowerCase()  
);  
var capitalize = cacheStringFunction((str) => {  
 return str.charAt(0).toUpperCase() + str.slice(1);  
});  
var toHandlerKey = cacheStringFunction((str) => {  
 const s = str ? `on${capitalize(str)}` : ``;  
 return s;  
});  
var hasChanged = (value, oldValue) => !Object.is(value, oldValue);  
var invokeArrayFns = (fns, ...arg) => {  
 for (let i = 0; i < fns.length; i++) {  
 fns[i](...arg);  
 }  
};  
var def = (obj, key, value, writable = false) => {  
 Object.defineProperty(obj, key, {  
 configurable: true,  
 enumerable: false,  
 writable,  
 value  
 });  
};  
var looseToNumber = (val) => {  
 const n = parseFloat(val);  
 return isNaN(n) ? val : n;  
};  
var toNumber = (val) => {  
 const n = isString(val) ? Number(val) : NaN;  
 return isNaN(n) ? val : n;  
};  
var \_globalThis;  
var getGlobalThis = () => {  
 return \_globalThis || (\_globalThis = typeof globalThis !== "undefined" ? globalThis : typeof self !== "undefined" ? self : typeof window !== "undefined" ? window : typeof global !== "undefined" ? global : {});  
};  
var GLOBALS\_ALLOWED = "Infinity,undefined,NaN,isFinite,isNaN,parseFloat,parseInt,decodeURI,decodeURIComponent,encodeURI,encodeURIComponent,Math,Number,Date,Array,Object,Boolean,String,RegExp,Map,Set,JSON,Intl,BigInt,console,Error";  
var isGloballyAllowed = makeMap(GLOBALS\_ALLOWED);  
function normalizeStyle(value) {  
 if (isArray(value)) {  
 const res = {};  
 for (let i = 0; i < value.length; i++) {  
 const item = value[i];  
 const normalized = isString(item) ? parseStringStyle(item) : normalizeStyle(item);  
 if (normalized) {  
 for (const key in normalized) {  
 res[key] = normalized[key];  
 }  
 }  
 }  
 return res;  
 } else if (isString(value) || isObject(value)) {  
 return value;  
 }  
}  
var listDelimiterRE = /;(?![^(]\*\))/g;  
var propertyDelimiterRE = /:([^]+)/;  
var styleCommentRE = /\/\\*[^]\*?\\*\//g;  
function parseStringStyle(cssText) {  
 const ret = {};  
 cssText.replace(styleCommentRE, "").split(listDelimiterRE).forEach((item) => {  
 if (item) {  
 const tmp = item.split(propertyDelimiterRE);  
 tmp.length > 1 && (ret[tmp[0].trim()] = tmp[1].trim());  
 }  
 });  
 return ret;  
}  
function stringifyStyle(styles) {  
 let ret = "";  
 if (!styles || isString(styles)) {  
 return ret;  
 }  
 for (const key in styles) {  
 const value = styles[key];  
 if (isString(value) || typeof value === "number") {  
 const normalizedKey = key.startsWith(`--`) ? key : hyphenate(key);  
 ret += `${normalizedKey}:${value};`;  
 }  
 }  
 return ret;  
}  
function normalizeClass(value) {  
 let res = "";  
 if (isString(value)) {  
 res = value;  
 } else if (isArray(value)) {  
 for (let i = 0; i < value.length; i++) {  
 const normalized = normalizeClass(value[i]);  
 if (normalized) {  
 res += normalized + " ";  
 }  
 }  
 } else if (isObject(value)) {  
 for (const name in value) {  
 if (value[name]) {  
 res += name + " ";  
 }  
 }  
 }  
 return res.trim();  
}  
function normalizeProps(props) {  
 if (!props) return null;  
 let { class: klass, style } = props;  
 if (klass && !isString(klass)) {  
 props.class = normalizeClass(klass);  
 }  
 if (style) {  
 props.style = normalizeStyle(style);  
 }  
 return props;  
}  
var HTML\_TAGS = "html,body,base,head,link,meta,style,title,address,article,aside,footer,header,hgroup,h1,h2,h3,h4,h5,h6,nav,section,div,dd,dl,dt,figcaption,figure,picture,hr,img,li,main,ol,p,pre,ul,a,b,abbr,bdi,bdo,br,cite,code,data,dfn,em,i,kbd,mark,q,rp,rt,ruby,s,samp,small,span,strong,sub,sup,time,u,var,wbr,area,audio,map,track,video,embed,object,param,source,canvas,script,noscript,del,ins,caption,col,colgroup,table,thead,tbody,td,th,tr,button,datalist,fieldset,form,input,label,legend,meter,optgroup,option,output,progress,select,textarea,details,dialog,menu,summary,template,blockquote,iframe,tfoot";  
var SVG\_TAGS = "svg,animate,animateMotion,animateTransform,circle,clipPath,color-profile,defs,desc,discard,ellipse,feBlend,feColorMatrix,feComponentTransfer,feComposite,feConvolveMatrix,feDiffuseLighting,feDisplacementMap,feDistantLight,feDropShadow,feFlood,feFuncA,feFuncB,feFuncG,feFuncR,feGaussianBlur,feImage,feMerge,feMergeNode,feMorphology,feOffset,fePointLight,feSpecularLighting,feSpotLight,feTile,feTurbulence,filter,foreignObject,g,hatch,hatchpath,image,line,linearGradient,marker,mask,mesh,meshgradient,meshpatch,meshrow,metadata,mpath,path,pattern,polygon,polyline,radialGradient,rect,set,solidcolor,stop,switch,symbol,text,textPath,title,tspan,unknown,use,view";  
var MATH\_TAGS = "annotation,annotation-xml,maction,maligngroup,malignmark,math,menclose,merror,mfenced,mfrac,mfraction,mglyph,mi,mlabeledtr,mlongdiv,mmultiscripts,mn,mo,mover,mpadded,mphantom,mprescripts,mroot,mrow,ms,mscarries,mscarry,msgroup,msline,mspace,msqrt,msrow,mstack,mstyle,msub,msubsup,msup,mtable,mtd,mtext,mtr,munder,munderover,none,semantics";  
var VOID\_TAGS = "area,base,br,col,embed,hr,img,input,link,meta,param,source,track,wbr";  
var isHTMLTag = makeMap(HTML\_TAGS);  
var isSVGTag = makeMap(SVG\_TAGS);  
var isMathMLTag = makeMap(MATH\_TAGS);  
var isVoidTag = makeMap(VOID\_TAGS);  
var specialBooleanAttrs = `itemscope,allowfullscreen,formnovalidate,ismap,nomodule,novalidate,readonly`;  
var isSpecialBooleanAttr = makeMap(specialBooleanAttrs);  
var isBooleanAttr = makeMap(  
 specialBooleanAttrs + `,async,autofocus,autoplay,controls,default,defer,disabled,hidden,inert,loop,open,required,reversed,scoped,seamless,checked,muted,multiple,selected`  
);  
function includeBooleanAttr(value) {  
 return !!value || value === "";  
}  
var isKnownHtmlAttr = makeMap(  
 `accept,accept-charset,accesskey,action,align,allow,alt,async,autocapitalize,autocomplete,autofocus,autoplay,background,bgcolor,border,buffered,capture,challenge,charset,checked,cite,class,code,codebase,color,cols,colspan,content,contenteditable,contextmenu,controls,coords,crossorigin,csp,data,datetime,decoding,default,defer,dir,dirname,disabled,download,draggable,dropzone,enctype,enterkeyhint,for,form,formaction,formenctype,formmethod,formnovalidate,formtarget,headers,height,hidden,high,href,hreflang,http-equiv,icon,id,importance,inert,integrity,ismap,itemprop,keytype,kind,label,lang,language,loading,list,loop,low,manifest,max,maxlength,minlength,media,min,multiple,muted,name,novalidate,open,optimum,pattern,ping,placeholder,poster,preload,radiogroup,readonly,referrerpolicy,rel,required,reversed,rows,rowspan,sandbox,scope,scoped,selected,shape,size,sizes,slot,span,spellcheck,src,srcdoc,srclang,srcset,start,step,style,summary,tabindex,target,title,translate,type,usemap,value,width,wrap`  
);  
var isKnownSvgAttr = makeMap(  
 `xmlns,accent-height,accumulate,additive,alignment-baseline,alphabetic,amplitude,arabic-form,ascent,attributeName,attributeType,azimuth,baseFrequency,baseline-shift,baseProfile,bbox,begin,bias,by,calcMode,cap-height,class,clip,clipPathUnits,clip-path,clip-rule,color,color-interpolation,color-interpolation-filters,color-profile,color-rendering,contentScriptType,contentStyleType,crossorigin,cursor,cx,cy,d,decelerate,descent,diffuseConstant,direction,display,divisor,dominant-baseline,dur,dx,dy,edgeMode,elevation,enable-background,end,exponent,fill,fill-opacity,fill-rule,filter,filterRes,filterUnits,flood-color,flood-opacity,font-family,font-size,font-size-adjust,font-stretch,font-style,font-variant,font-weight,format,from,fr,fx,fy,g1,g2,glyph-name,glyph-orientation-horizontal,glyph-orientation-vertical,glyphRef,gradientTransform,gradientUnits,hanging,height,href,hreflang,horiz-adv-x,horiz-origin-x,id,ideographic,image-rendering,in,in2,intercept,k,k1,k2,k3,k4,kernelMatrix,kernelUnitLength,kerning,keyPoints,keySplines,keyTimes,lang,lengthAdjust,letter-spacing,lighting-color,limitingConeAngle,local,marker-end,marker-mid,marker-start,markerHeight,markerUnits,markerWidth,mask,maskContentUnits,maskUnits,mathematical,max,media,method,min,mode,name,numOctaves,offset,opacity,operator,order,orient,orientation,origin,overflow,overline-position,overline-thickness,panose-1,paint-order,path,pathLength,patternContentUnits,patternTransform,patternUnits,ping,pointer-events,points,pointsAtX,pointsAtY,pointsAtZ,preserveAlpha,preserveAspectRatio,primitiveUnits,r,radius,referrerPolicy,refX,refY,rel,rendering-intent,repeatCount,repeatDur,requiredExtensions,requiredFeatures,restart,result,rotate,rx,ry,scale,seed,shape-rendering,slope,spacing,specularConstant,specularExponent,speed,spreadMethod,startOffset,stdDeviation,stemh,stemv,stitchTiles,stop-color,stop-opacity,strikethrough-position,strikethrough-thickness,string,stroke,stroke-dasharray,stroke-dashoffset,stroke-linecap,stroke-linejoin,stroke-miterlimit,stroke-opacity,stroke-width,style,surfaceScale,systemLanguage,tabindex,tableValues,target,targetX,targetY,text-anchor,text-decoration,text-rendering,textLength,to,transform,transform-origin,type,u1,u2,underline-position,underline-thickness,unicode,unicode-bidi,unicode-range,units-per-em,v-alphabetic,v-hanging,v-ideographic,v-mathematical,values,vector-effect,version,vert-adv-y,vert-origin-x,vert-origin-y,viewBox,viewTarget,visibility,width,widths,word-spacing,writing-mode,x,x-height,x1,x2,xChannelSelector,xlink:actuate,xlink:arcrole,xlink:href,xlink:role,xlink:show,xlink:title,xlink:type,xmlns:xlink,xml:base,xml:lang,xml:space,y,y1,y2,yChannelSelector,z,zoomAndPan`  
);  
function isRenderableAttrValue(value) {  
 if (value == null) {  
 return false;  
 }  
 const type = typeof value;  
 return type === "string" || type === "number" || type === "boolean";  
}  
function looseCompareArrays(a, b) {  
 if (a.length !== b.length) return false;  
 let equal = true;  
 for (let i = 0; equal && i < a.length; i++) {  
 equal = looseEqual(a[i], b[i]);  
 }  
 return equal;  
}  
function looseEqual(a, b) {  
 if (a === b) return true;  
 let aValidType = isDate(a);  
 let bValidType = isDate(b);  
 if (aValidType || bValidType) {  
 return aValidType && bValidType ? a.getTime() === b.getTime() : false;  
 }  
 aValidType = isSymbol(a);  
 bValidType = isSymbol(b);  
 if (aValidType || bValidType) {  
 return a === b;  
 }  
 aValidType = isArray(a);  
 bValidType = isArray(b);  
 if (aValidType || bValidType) {  
 return aValidType && bValidType ? looseCompareArrays(a, b) : false;  
 }  
 aValidType = isObject(a);  
 bValidType = isObject(b);  
 if (aValidType || bValidType) {  
 if (!aValidType || !bValidType) {  
 return false;  
 }  
 const aKeysCount = Object.keys(a).length;  
 const bKeysCount = Object.keys(b).length;  
 if (aKeysCount !== bKeysCount) {  
 return false;  
 }  
 for (const key in a) {  
 const aHasKey = a.hasOwnProperty(key);  
 const bHasKey = b.hasOwnProperty(key);  
 if (aHasKey && !bHasKey || !aHasKey && bHasKey || !looseEqual(a[key], b[key])) {  
 return false;  
 }  
 }  
 }  
 return String(a) === String(b);  
}  
function looseIndexOf(arr, val) {  
 return arr.findIndex((item) => looseEqual(item, val));  
}  
var isRef = (val) => {  
 return !!(val && val.\_\_v\_isRef === true);  
};  
var toDisplayString = (val) => {  
 return isString(val) ? val : val == null ? "" : isArray(val) || isObject(val) && (val.toString === objectToString || !isFunction(val.toString)) ? isRef(val) ? toDisplayString(val.value) : JSON.stringify(val, replacer, 2) : String(val);  
};  
var replacer = (\_key, val) => {  
 if (isRef(val)) {  
 return replacer(\_key, val.value);  
 } else if (isMap(val)) {  
 return {  
 [`Map(${val.size})`]: [...val.entries()].reduce(  
 (entries, [key, val2], i) => {  
 entries[stringifySymbol(key, i) + " =>"] = val2;  
 return entries;  
 },  
 {}  
 )  
 };  
 } else if (isSet(val)) {  
 return {  
 [`Set(${val.size})`]: [...val.values()].map((v) => stringifySymbol(v))  
 };  
 } else if (isSymbol(val)) {  
 return stringifySymbol(val);  
 } else if (isObject(val) && !isArray(val) && !isPlainObject(val)) {  
 return String(val);  
 }  
 return val;  
};  
var stringifySymbol = (v, i = "") => {  
 var \_a;  
 return (  
 isSymbol(v) ? `Symbol(${(\_a = v.description) != null ? \_a : i})` : v  
 );  
};  
function warn(msg, ...args) {  
 console.warn(`[Vue warn] ${msg}`, ...args);  
}  
var activeEffectScope;  
var EffectScope = class {  
 constructor(detached = false) {  
 this.detached = detached;  
 this.\_active = true;  
 this.effects = [];  
 this.cleanups = [];  
 this.parent = activeEffectScope;  
 if (!detached && activeEffectScope) {  
 this.index = (activeEffectScope.scopes || (activeEffectScope.scopes = [])).push(  
 this  
 ) - 1;  
 }  
 }  
 get active() {  
 return this.\_active;  
 }  
 run(fn) {  
 if (this.\_active) {  
 const currentEffectScope = activeEffectScope;  
 try {  
 activeEffectScope = this;  
 return fn();  
 } finally {  
 activeEffectScope = currentEffectScope;  
 }  
 } else if (true) {  
 warn(`cannot run an inactive effect scope.`);  
 }  
 }  
 on() {  
 activeEffectScope = this;  
 }  
 off() {  
 activeEffectScope = this.parent;  
 }  
 stop(fromParent) {  
 if (this.\_active) {  
 let i, l;  
 for (i = 0, l = this.effects.length; i < l; i++) {  
 this.effects[i].stop();  
 }  
 for (i = 0, l = this.cleanups.length; i < l; i++) {  
 this.cleanups[i]();  
 }  
 if (this.scopes) {  
 for (i = 0, l = this.scopes.length; i < l; i++) {  
 this.scopes[i].stop(true);  
 }  
 }  
 if (!this.detached && this.parent && !fromParent) {  
 const last = this.parent.scopes.pop();  
 if (last && last !== this) {  
 this.parent.scopes[this.index] = last;  
 last.index = this.index;  
 }  
 }  
 this.parent = void 0;  
 this.\_active = false;  
 }  
 }  
};  
function effectScope(detached) {  
 return new EffectScope(detached);  
}  
function recordEffectScope(effect2, scope = activeEffectScope) {  
 if (scope && scope.active) {  
 scope.effects.push(effect2);  
 }  
}  
function getCurrentScope() {  
 return activeEffectScope;  
}  
function onScopeDispose(fn) {  
 if (activeEffectScope) {  
 activeEffectScope.cleanups.push(fn);  
 } else if (true) {  
 warn(  
 `onScopeDispose() is called when there is no active effect scope to be associated with.`  
 );  
 }  
}  
var activeEffect;  
var ReactiveEffect = class {  
 constructor(fn, trigger2, scheduler, scope) {  
 this.fn = fn;  
 this.trigger = trigger2;  
 this.scheduler = scheduler;  
 this.active = true;  
 this.deps = [];  
 this.\_dirtyLevel = 4;  
 this.\_trackId = 0;  
 this.\_runnings = 0;  
 this.\_shouldSchedule = false;  
 this.\_depsLength = 0;  
 recordEffectScope(this, scope);  
 }  
 get dirty() {  
 if (this.\_dirtyLevel === 2 || this.\_dirtyLevel === 3) {  
 this.\_dirtyLevel = 1;  
 pauseTracking();  
 for (let i = 0; i < this.\_depsLength; i++) {  
 const dep = this.deps[i];  
 if (dep.computed) {  
 triggerComputed(dep.computed);  
 if (this.\_dirtyLevel >= 4) {  
 break;  
 }  
 }  
 }  
 if (this.\_dirtyLevel === 1) {  
 this.\_dirtyLevel = 0;  
 }  
 resetTracking();  
 }  
 return this.\_dirtyLevel >= 4;  
 }  
 set dirty(v) {  
 this.\_dirtyLevel = v ? 4 : 0;  
 }  
 run() {  
 this.\_dirtyLevel = 0;  
 if (!this.active) {  
 return this.fn();  
 }  
 let lastShouldTrack = shouldTrack;  
 let lastEffect = activeEffect;  
 try {  
 shouldTrack = true;  
 activeEffect = this;  
 this.\_runnings++;  
 preCleanupEffect(this);  
 return this.fn();  
 } finally {  
 postCleanupEffect(this);  
 this.\_runnings--;  
 activeEffect = lastEffect;  
 shouldTrack = lastShouldTrack;  
 }  
 }  
 stop() {  
 if (this.active) {  
 preCleanupEffect(this);  
 postCleanupEffect(this);  
 this.onStop && this.onStop();  
 this.active = false;  
 }  
 }  
};  
function triggerComputed(computed3) {  
 return computed3.value;  
}  
function preCleanupEffect(effect2) {  
 effect2.\_trackId++;  
 effect2.\_depsLength = 0;  
}  
function postCleanupEffect(effect2) {  
 if (effect2.deps.length > effect2.\_depsLength) {  
 for (let i = effect2.\_depsLength; i < effect2.deps.length; i++) {  
 cleanupDepEffect(effect2.deps[i], effect2);  
 }  
 effect2.deps.length = effect2.\_depsLength;  
 }  
}  
function cleanupDepEffect(dep, effect2) {  
 const trackId = dep.get(effect2);  
 if (trackId !== void 0 && effect2.\_trackId !== trackId) {  
 dep.delete(effect2);  
 if (dep.size === 0) {  
 dep.cleanup();  
 }  
 }  
}  
function effect(fn, options) {  
 if (fn.effect instanceof ReactiveEffect) {  
 fn = fn.effect.fn;  
 }  
 const \_effect = new ReactiveEffect(fn, NOOP, () => {  
 if (\_effect.dirty) {  
 \_effect.run();  
 }  
 });  
 if (options) {  
 extend(\_effect, options);  
 if (options.scope) recordEffectScope(\_effect, options.scope);  
 }  
 if (!options || !options.lazy) {  
 \_effect.run();  
 }  
 const runner = \_effect.run.bind(\_effect);  
 runner.effect = \_effect;  
 return runner;  
}  
function stop(runner) {  
 runner.effect.stop();  
}  
var shouldTrack = true;  
var pauseScheduleStack = 0;  
var trackStack = [];  
function pauseTracking() {  
 trackStack.push(shouldTrack);  
 shouldTrack = false;  
}  
function resetTracking() {  
 const last = trackStack.pop();  
 shouldTrack = last === void 0 ? true : last;  
}  
function pauseScheduling() {  
 pauseScheduleStack++;  
}  
function resetScheduling() {  
 pauseScheduleStack--;  
 while (!pauseScheduleStack && queueEffectSchedulers.length) {  
 queueEffectSchedulers.shift()();  
 }  
}  
function trackEffect(effect2, dep, debuggerEventExtraInfo) {  
 var \_a;  
 if (dep.get(effect2) !== effect2.\_trackId) {  
 dep.set(effect2, effect2.\_trackId);  
 const oldDep = effect2.deps[effect2.\_depsLength];  
 if (oldDep !== dep) {  
 if (oldDep) {  
 cleanupDepEffect(oldDep, effect2);  
 }  
 effect2.deps[effect2.\_depsLength++] = dep;  
 } else {  
 effect2.\_depsLength++;  
 }  
 if (true) {  
 (\_a = effect2.onTrack) == null ? void 0 : \_a.call(effect2, extend({ effect: effect2 }, debuggerEventExtraInfo));  
 }  
 }  
}  
var queueEffectSchedulers = [];  
function triggerEffects(dep, dirtyLevel, debuggerEventExtraInfo) {  
 var \_a;  
 pauseScheduling();  
 for (const effect2 of dep.keys()) {  
 let tracking;  
 if (effect2.\_dirtyLevel < dirtyLevel && (tracking != null ? tracking : tracking = dep.get(effect2) === effect2.\_trackId)) {  
 effect2.\_shouldSchedule || (effect2.\_shouldSchedule = effect2.\_dirtyLevel === 0);  
 effect2.\_dirtyLevel = dirtyLevel;  
 }  
 if (effect2.\_shouldSchedule && (tracking != null ? tracking : tracking = dep.get(effect2) === effect2.\_trackId)) {  
 if (true) {  
 (\_a = effect2.onTrigger) == null ? void 0 : \_a.call(effect2, extend({ effect: effect2 }, debuggerEventExtraInfo));  
 }  
 effect2.trigger();  
 if ((!effect2.\_runnings || effect2.allowRecurse) && effect2.\_dirtyLevel !== 2) {  
 effect2.\_shouldSchedule = false;  
 if (effect2.scheduler) {  
 queueEffectSchedulers.push(effect2.scheduler);  
 }  
 }  
 }  
 }  
 resetScheduling();  
}  
var createDep = (cleanup, computed3) => {  
 const dep = /\* @\_\_PURE\_\_ \*/ new Map();  
 dep.cleanup = cleanup;  
 dep.computed = computed3;  
 return dep;  
};  
var targetMap = /\* @\_\_PURE\_\_ \*/ new WeakMap();  
var ITERATE\_KEY = Symbol(true ? "iterate" : "");  
var MAP\_KEY\_ITERATE\_KEY = Symbol(true ? "Map key iterate" : "");  
function track(target, type, key) {  
 if (shouldTrack && activeEffect) {  
 let depsMap = targetMap.get(target);  
 if (!depsMap) {  
 targetMap.set(target, depsMap = /\* @\_\_PURE\_\_ \*/ new Map());  
 }  
 let dep = depsMap.get(key);  
 if (!dep) {  
 depsMap.set(key, dep = createDep(() => depsMap.delete(key)));  
 }  
 trackEffect(  
 activeEffect,  
 dep,  
 true ? {  
 target,  
 type,  
 key  
 } : void 0  
 );  
 }  
}  
function trigger(target, type, key, newValue, oldValue, oldTarget) {  
 const depsMap = targetMap.get(target);  
 if (!depsMap) {  
 return;  
 }  
 let deps = [];  
 if (type === "clear") {  
 deps = [...depsMap.values()];  
 } else if (key === "length" && isArray(target)) {  
 const newLength = Number(newValue);  
 depsMap.forEach((dep, key2) => {  
 if (key2 === "length" || !isSymbol(key2) && key2 >= newLength) {  
 deps.push(dep);  
 }  
 });  
 } else {  
 if (key !== void 0) {  
 deps.push(depsMap.get(key));  
 }  
 switch (type) {  
 case "add":  
 if (!isArray(target)) {  
 deps.push(depsMap.get(ITERATE\_KEY));  
 if (isMap(target)) {  
 deps.push(depsMap.get(MAP\_KEY\_ITERATE\_KEY));  
 }  
 } else if (isIntegerKey(key)) {  
 deps.push(depsMap.get("length"));  
 }  
 break;  
 case "delete":  
 if (!isArray(target)) {  
 deps.push(depsMap.get(ITERATE\_KEY));  
 if (isMap(target)) {  
 deps.push(depsMap.get(MAP\_KEY\_ITERATE\_KEY));  
 }  
 }  
 break;  
 case "set":  
 if (isMap(target)) {  
 deps.push(depsMap.get(ITERATE\_KEY));  
 }  
 break;  
 }  
 }  
 pauseScheduling();  
 for (const dep of deps) {  
 if (dep) {  
 triggerEffects(  
 dep,  
 4,  
 true ? {  
 target,  
 type,  
 key,  
 newValue,  
 oldValue,  
 oldTarget  
 } : void 0  
 );  
 }  
 }  
 resetScheduling();  
}  
function getDepFromReactive(object, key) {  
 const depsMap = targetMap.get(object);  
 return depsMap && depsMap.get(key);  
}  
var isNonTrackableKeys = makeMap(`\_\_proto\_\_,\_\_v\_isRef,\_\_isVue`);  
var builtInSymbols = new Set(  
 Object.getOwnPropertyNames(Symbol).filter((key) => key !== "arguments" && key !== "caller").map((key) => Symbol[key]).filter(isSymbol)  
);  
var arrayInstrumentations = createArrayInstrumentations();  
function createArrayInstrumentations() {  
 const instrumentations = {};  
 ["includes", "indexOf", "lastIndexOf"].forEach((key) => {  
 instrumentations[key] = function(...args) {  
 const arr = toRaw(this);  
 for (let i = 0, l = this.length; i < l; i++) {  
 track(arr, "get", i + "");  
 }  
 const res = arr[key](...args);  
 if (res === -1 || res === false) {  
 return arr[key](...args.map(toRaw));  
 } else {  
 return res;  
 }  
 };  
 });  
 ["push", "pop", "shift", "unshift", "splice"].forEach((key) => {  
 instrumentations[key] = function(...args) {  
 pauseTracking();  
 pauseScheduling();  
 const res = toRaw(this)[key].apply(this, args);  
 resetScheduling();  
 resetTracking();  
 return res;  
 };  
 });  
 return instrumentations;  
}  
function hasOwnProperty2(key) {  
 if (!isSymbol(key)) key = String(key);  
 const obj = toRaw(this);  
 track(obj, "has", key);  
 return obj.hasOwnProperty(key);  
}  
var BaseReactiveHandler = class {  
 constructor(\_isReadonly = false, \_isShallow = false) {  
 this.\_isReadonly = \_isReadonly;  
 this.\_isShallow = \_isShallow;  
 }  
 get(target, key, receiver) {  
 const isReadonly2 = this.\_isReadonly, isShallow2 = this.\_isShallow;  
 if (key === "\_\_v\_isReactive") {  
 return !isReadonly2;  
 } else if (key === "\_\_v\_isReadonly") {  
 return isReadonly2;  
 } else if (key === "\_\_v\_isShallow") {  
 return isShallow2;  
 } else if (key === "\_\_v\_raw") {  
 if (receiver === (isReadonly2 ? isShallow2 ? shallowReadonlyMap : readonlyMap : isShallow2 ? shallowReactiveMap : reactiveMap).get(target) || // receiver is not the reactive proxy, but has the same prototype  
 Object.getPrototypeOf(target) === Object.getPrototypeOf(receiver)) {  
 return target;  
 }  
 return;  
 }  
 const targetIsArray = isArray(target);  
 if (!isReadonly2) {  
 if (targetIsArray && hasOwn(arrayInstrumentations, key)) {  
 return Reflect.get(arrayInstrumentations, key, receiver);  
 }  
 if (key === "hasOwnProperty") {  
 return hasOwnProperty2;  
 }  
 }  
 const res = Reflect.get(target, key, receiver);  
 if (isSymbol(key) ? builtInSymbols.has(key) : isNonTrackableKeys(key)) {  
 return res;  
 }  
 if (!isReadonly2) {  
 track(target, "get", key);  
 }  
 if (isShallow2) {  
 return res;  
 }  
 if (isRef2(res)) {  
 return targetIsArray && isIntegerKey(key) ? res : res.value;  
 }  
 if (isObject(res)) {  
 return isReadonly2 ? readonly(res) : reactive(res);  
 }  
 return res;  
 }  
};  
var MutableReactiveHandler = class extends BaseReactiveHandler {  
 constructor(isShallow2 = false) {  
 super(false, isShallow2);  
 }  
 set(target, key, value, receiver) {  
 let oldValue = target[key];  
 if (!this.\_isShallow) {  
 const isOldValueReadonly = isReadonly(oldValue);  
 if (!isShallow(value) && !isReadonly(value)) {  
 oldValue = toRaw(oldValue);  
 value = toRaw(value);  
 }  
 if (!isArray(target) && isRef2(oldValue) && !isRef2(value)) {  
 if (isOldValueReadonly) {  
 return false;  
 } else {  
 oldValue.value = value;  
 return true;  
 }  
 }  
 }  
 const hadKey = isArray(target) && isIntegerKey(key) ? Number(key) < target.length : hasOwn(target, key);  
 const result = Reflect.set(target, key, value, receiver);  
 if (target === toRaw(receiver)) {  
 if (!hadKey) {  
 trigger(target, "add", key, value);  
 } else if (hasChanged(value, oldValue)) {  
 trigger(target, "set", key, value, oldValue);  
 }  
 }  
 return result;  
 }  
 deleteProperty(target, key) {  
 const hadKey = hasOwn(target, key);  
 const oldValue = target[key];  
 const result = Reflect.deleteProperty(target, key);  
 if (result && hadKey) {  
 trigger(target, "delete", key, void 0, oldValue);  
 }  
 return result;  
 }  
 has(target, key) {  
 const result = Reflect.has(target, key);  
 if (!isSymbol(key) || !builtInSymbols.has(key)) {  
 track(target, "has", key);  
 }  
 return result;  
 }  
 ownKeys(target) {  
 track(  
 target,  
 "iterate",  
 isArray(target) ? "length" : ITERATE\_KEY  
 );  
 return Reflect.ownKeys(target);  
 }  
};  
var ReadonlyReactiveHandler = class extends BaseReactiveHandler {  
 constructor(isShallow2 = false) {  
 super(true, isShallow2);  
 }  
 set(target, key) {  
 if (true) {  
 warn(  
 `Set operation on key "${String(key)}" failed: target is readonly.`,  
 target  
 );  
 }  
 return true;  
 }  
 deleteProperty(target, key) {  
 if (true) {  
 warn(  
 `Delete operation on key "${String(key)}" failed: target is readonly.`,  
 target  
 );  
 }  
 return true;  
 }  
};  
var mutableHandlers = new MutableReactiveHandler();  
var readonlyHandlers = new ReadonlyReactiveHandler();  
var shallowReactiveHandlers = new MutableReactiveHandler(  
 true  
);  
var shallowReadonlyHandlers = new ReadonlyReactiveHandler(true);  
var toShallow = (value) => value;  
var getProto = (v) => Reflect.getPrototypeOf(v);  
function get(target, key, isReadonly2 = false, isShallow2 = false) {  
 target = target["\_\_v\_raw"];  
 const rawTarget = toRaw(target);  
 const rawKey = toRaw(key);  
 if (!isReadonly2) {  
 if (hasChanged(key, rawKey)) {  
 track(rawTarget, "get", key);  
 }  
 track(rawTarget, "get", rawKey);  
 }  
 const { has: has2 } = getProto(rawTarget);  
 const wrap = isShallow2 ? toShallow : isReadonly2 ? toReadonly : toReactive;  
 if (has2.call(rawTarget, key)) {  
 return wrap(target.get(key));  
 } else if (has2.call(rawTarget, rawKey)) {  
 return wrap(target.get(rawKey));  
 } else if (target !== rawTarget) {  
 target.get(key);  
 }  
}  
function has(key, isReadonly2 = false) {  
 const target = this["\_\_v\_raw"];  
 const rawTarget = toRaw(target);  
 const rawKey = toRaw(key);  
 if (!isReadonly2) {  
 if (hasChanged(key, rawKey)) {  
 track(rawTarget, "has", key);  
 }  
 track(rawTarget, "has", rawKey);  
 }  
 return key === rawKey ? target.has(key) : target.has(key) || target.has(rawKey);  
}  
function size(target, isReadonly2 = false) {  
 target = target["\_\_v\_raw"];  
 !isReadonly2 && track(toRaw(target), "iterate", ITERATE\_KEY);  
 return Reflect.get(target, "size", target);  
}  
function add(value, \_isShallow = false) {  
 if (!\_isShallow && !isShallow(value) && !isReadonly(value)) {  
 value = toRaw(value);  
 }  
 const target = toRaw(this);  
 const proto = getProto(target);  
 const hadKey = proto.has.call(target, value);  
 if (!hadKey) {  
 target.add(value);  
 trigger(target, "add", value, value);  
 }  
 return this;  
}  
function set(key, value, \_isShallow = false) {  
 if (!\_isShallow && !isShallow(value) && !isReadonly(value)) {  
 value = toRaw(value);  
 }  
 const target = toRaw(this);  
 const { has: has2, get: get2 } = getProto(target);  
 let hadKey = has2.call(target, key);  
 if (!hadKey) {  
 key = toRaw(key);  
 hadKey = has2.call(target, key);  
 } else if (true) {  
 checkIdentityKeys(target, has2, key);  
 }  
 const oldValue = get2.call(target, key);  
 target.set(key, value);  
 if (!hadKey) {  
 trigger(target, "add", key, value);  
 } else if (hasChanged(value, oldValue)) {  
 trigger(target, "set", key, value, oldValue);  
 }  
 return this;  
}  
function deleteEntry(key) {  
 const target = toRaw(this);  
 const { has: has2, get: get2 } = getProto(target);  
 let hadKey = has2.call(target, key);  
 if (!hadKey) {  
 key = toRaw(key);  
 hadKey = has2.call(target, key);  
 } else if (true) {  
 checkIdentityKeys(target, has2, key);  
 }  
 const oldValue = get2 ? get2.call(target, key) : void 0;  
 const result = target.delete(key);  
 if (hadKey) {  
 trigger(target, "delete", key, void 0, oldValue);  
 }  
 return result;  
}  
function clear() {  
 const target = toRaw(this);  
 const hadItems = target.size !== 0;  
 const oldTarget = true ? isMap(target) ? new Map(target) : new Set(target) : void 0;  
 const result = target.clear();  
 if (hadItems) {  
 trigger(target, "clear", void 0, void 0, oldTarget);  
 }  
 return result;  
}  
function createForEach(isReadonly2, isShallow2) {  
 return function forEach(callback, thisArg) {  
 const observed = this;  
 const target = observed["\_\_v\_raw"];  
 const rawTarget = toRaw(target);  
 const wrap = isShallow2 ? toShallow : isReadonly2 ? toReadonly : toReactive;  
 !isReadonly2 && track(rawTarget, "iterate", ITERATE\_KEY);  
 return target.forEach((value, key) => {  
 return callback.call(thisArg, wrap(value), wrap(key), observed);  
 });  
 };  
}  
function createIterableMethod(method, isReadonly2, isShallow2) {  
 return function(...args) {  
 const target = this["\_\_v\_raw"];  
 const rawTarget = toRaw(target);  
 const targetIsMap = isMap(rawTarget);  
 const isPair = method === "entries" || method === Symbol.iterator && targetIsMap;  
 const isKeyOnly = method === "keys" && targetIsMap;  
 const innerIterator = target[method](...args);  
 const wrap = isShallow2 ? toShallow : isReadonly2 ? toReadonly : toReactive;  
 !isReadonly2 && track(  
 rawTarget,  
 "iterate",  
 isKeyOnly ? MAP\_KEY\_ITERATE\_KEY : ITERATE\_KEY  
 );  
 return {  
 next() {  
 const { value, done } = innerIterator.next();  
 return done ? { value, done } : {  
 value: isPair ? [wrap(value[0]), wrap(value[1])] : wrap(value),  
 done  
 };  
 },  
 [Symbol.iterator]() {  
 return this;  
 }  
 };  
 };  
}  
function createReadonlyMethod(type) {  
 return function(...args) {  
 if (true) {  
 const key = args[0] ? `on key "${args[0]}" ` : ``;  
 warn(  
 `${capitalize(type)} operation ${key}failed: target is readonly.`,  
 toRaw(this)  
 );  
 }  
 return type === "delete" ? false : type === "clear" ? void 0 : this;  
 };  
}  
function createInstrumentations() {  
 const mutableInstrumentations2 = {  
 get(key) {  
 return get(this, key);  
 },  
 get size() {  
 return size(this);  
 },  
 has,  
 add,  
 set,  
 delete: deleteEntry,  
 clear,  
 forEach: createForEach(false, false)  
 };  
 const shallowInstrumentations2 = {  
 get(key) {  
 return get(this, key, false, true);  
 },  
 get size() {  
 return size(this);  
 },  
 has,  
 add(value) {  
 return add.call(this, value, true);  
 },  
 set(key, value) {  
 return set.call(this, key, value, true);  
 },  
 delete: deleteEntry,  
 clear,  
 forEach: createForEach(false, true)  
 };  
 const readonlyInstrumentations2 = {  
 get(key) {  
 return get(this, key, true);  
 },  
 get size() {  
 return size(this, true);  
 },  
 has(key) {  
 return has.call(this, key, true);  
 },  
 add: createReadonlyMethod("add"),  
 set: createReadonlyMethod("set"),  
 delete: createReadonlyMethod("delete"),  
 clear: createReadonlyMethod("clear"),  
 forEach: createForEach(true, false)  
 };  
 const shallowReadonlyInstrumentations2 = {  
 get(key) {  
 return get(this, key, true, true);  
 },  
 get size() {  
 return size(this, true);  
 },  
 has(key) {  
 return has.call(this, key, true);  
 },  
 add: createReadonlyMethod("add"),  
 set: createReadonlyMethod("set"),  
 delete: createReadonlyMethod("delete"),  
 clear: createReadonlyMethod("clear"),  
 forEach: createForEach(true, true)  
 };  
 const iteratorMethods = [  
 "keys",  
 "values",  
 "entries",  
 Symbol.iterator  
 ];  
 iteratorMethods.forEach((method) => {  
 mutableInstrumentations2[method] = createIterableMethod(method, false, false);  
 readonlyInstrumentations2[method] = createIterableMethod(method, true, false);  
 shallowInstrumentations2[method] = createIterableMethod(method, false, true);  
 shallowReadonlyInstrumentations2[method] = createIterableMethod(  
 method,  
 true,  
 true  
 );  
 });  
 return [  
 mutableInstrumentations2,  
 readonlyInstrumentations2,  
 shallowInstrumentations2,  
 shallowReadonlyInstrumentations2  
 ];  
}  
var [  
 mutableInstrumentations,  
 readonlyInstrumentations,  
 shallowInstrumentations,  
 shallowReadonlyInstrumentations  
] = createInstrumentations();  
function createInstrumentationGetter(isReadonly2, shallow) {  
 const instrumentations = shallow ? isReadonly2 ? shallowReadonlyInstrumentations : shallowInstrumentations : isReadonly2 ? readonlyInstrumentations : mutableInstrumentations;  
 return (target, key, receiver) => {  
 if (key === "\_\_v\_isReactive") {  
 return !isReadonly2;  
 } else if (key === "\_\_v\_isReadonly") {  
 return isReadonly2;  
 } else if (key === "\_\_v\_raw") {  
 return target;  
 }  
 return Reflect.get(  
 hasOwn(instrumentations, key) && key in target ? instrumentations : target,  
 key,  
 receiver  
 );  
 };  
}  
var mutableCollectionHandlers = {  
 get: createInstrumentationGetter(false, false)  
};  
var shallowCollectionHandlers = {  
 get: createInstrumentationGetter(false, true)  
};  
var readonlyCollectionHandlers = {  
 get: createInstrumentationGetter(true, false)  
};  
var shallowReadonlyCollectionHandlers = {  
 get: createInstrumentationGetter(true, true)  
};  
function checkIdentityKeys(target, has2, key) {  
 const rawKey = toRaw(key);  
 if (rawKey !== key && has2.call(target, rawKey)) {  
 const type = toRawType(target);  
 warn(  
 `Reactive ${type} contains both the raw and reactive versions of the same object${type === `Map` ? ` as keys` : ``}, which can lead to inconsistencies. Avoid differentiating between the raw and reactive versions of an object and only use the reactive version if possible.`  
 );  
 }  
}  
var reactiveMap = /\* @\_\_PURE\_\_ \*/ new WeakMap();  
var shallowReactiveMap = /\* @\_\_PURE\_\_ \*/ new WeakMap();  
var readonlyMap = /\* @\_\_PURE\_\_ \*/ new WeakMap();  
var shallowReadonlyMap = /\* @\_\_PURE\_\_ \*/ new WeakMap();  
function targetTypeMap(rawType) {  
 switch (rawType) {  
 case "Object":  
 case "Array":  
 return 1;  
 case "Map":  
 case "Set":  
 case "WeakMap":  
 case "WeakSet":  
 return 2;  
 default:  
 return 0;  
 }  
}  
function getTargetType(value) {  
 return value["\_\_v\_skip"] || !Object.isExtensible(value) ? 0 : targetTypeMap(toRawType(value));  
}  
function reactive(target) {  
 if (isReadonly(target)) {  
 return target;  
 }  
 return createReactiveObject(  
 target,  
 false,  
 mutableHandlers,  
 mutableCollectionHandlers,  
 reactiveMap  
 );  
}  
function shallowReactive(target) {  
 return createReactiveObject(  
 target,  
 false,  
 shallowReactiveHandlers,  
 shallowCollectionHandlers,  
 shallowReactiveMap  
 );  
}  
function readonly(target) {  
 return createReactiveObject(  
 target,  
 true,  
 readonlyHandlers,  
 readonlyCollectionHandlers,  
 readonlyMap  
 );  
}  
function shallowReadonly(target) {  
 return createReactiveObject(  
 target,  
 true,  
 shallowReadonlyHandlers,  
 shallowReadonlyCollectionHandlers,  
 shallowReadonlyMap  
 );  
}  
function createReactiveObject(target, isReadonly2, baseHandlers, collectionHandlers, proxyMap) {  
 if (!isObject(target)) {  
 if (true) {  
 warn(  
 `value cannot be made ${isReadonly2 ? "readonly" : "reactive"}: ${String(  
 target  
 )}`  
 );  
 }  
 return target;  
 }  
 if (target["\_\_v\_raw"] && !(isReadonly2 && target["\_\_v\_isReactive"])) {  
 return target;  
 }  
 const existingProxy = proxyMap.get(target);  
 if (existingProxy) {  
 return existingProxy;  
 }  
 const targetType = getTargetType(target);  
 if (targetType === 0) {  
 return target;  
 }  
 const proxy = new Proxy(  
 target,  
 targetType === 2 ? collectionHandlers : baseHandlers  
 );  
 proxyMap.set(target, proxy);  
 return proxy;  
}  
function isReactive(value) {  
 if (isReadonly(value)) {  
 return isReactive(value["\_\_v\_raw"]);  
 }  
 return !!(value && value["\_\_v\_isReactive"]);  
}  
function isReadonly(value) {  
 return !!(value && value["\_\_v\_isReadonly"]);  
}  
function isShallow(value) {  
 return !!(value && value["\_\_v\_isShallow"]);  
}  
function isProxy(value) {  
 return value ? !!value["\_\_v\_raw"] : false;  
}  
function toRaw(observed) {  
 const raw = observed && observed["\_\_v\_raw"];  
 return raw ? toRaw(raw) : observed;  
}  
function markRaw(value) {  
 if (Object.isExtensible(value)) {  
 def(value, "\_\_v\_skip", true);  
 }  
 return value;  
}  
var toReactive = (value) => isObject(value) ? reactive(value) : value;  
var toReadonly = (value) => isObject(value) ? readonly(value) : value;  
var COMPUTED\_SIDE\_EFFECT\_WARN = `Computed is still dirty after getter evaluation, likely because a computed is mutating its own dependency in its getter. State mutations in computed getters should be avoided. Check the docs for more details: https://vuejs.org/guide/essentials/computed.html#getters-should-be-side-effect-free`;  
var ComputedRefImpl = class {  
 constructor(getter, \_setter, isReadonly2, isSSR) {  
 this.getter = getter;  
 this.\_setter = \_setter;  
 this.dep = void 0;  
 this.\_\_v\_isRef = true;  
 this["\_\_v\_isReadonly"] = false;  
 this.effect = new ReactiveEffect(  
 () => getter(this.\_value),  
 () => triggerRefValue(  
 this,  
 this.effect.\_dirtyLevel === 2 ? 2 : 3  
 )  
 );  
 this.effect.computed = this;  
 this.effect.active = this.\_cacheable = !isSSR;  
 this["\_\_v\_isReadonly"] = isReadonly2;  
 }  
 get value() {  
 const self2 = toRaw(this);  
 if ((!self2.\_cacheable || self2.effect.dirty) && hasChanged(self2.\_value, self2.\_value = self2.effect.run())) {  
 triggerRefValue(self2, 4);  
 }  
 trackRefValue(self2);  
 if (self2.effect.\_dirtyLevel >= 2) {  
 if (this.\_warnRecursive) {  
 warn(COMPUTED\_SIDE\_EFFECT\_WARN, `  
getter: `, this.getter);  
 }  
 triggerRefValue(self2, 2);  
 }  
 return self2.\_value;  
 }  
 set value(newValue) {  
 this.\_setter(newValue);  
 }  
 get \_dirty() {  
 return this.effect.dirty;  
 }  
 set \_dirty(v) {  
 this.effect.dirty = v;  
 }  
};  
function computed(getterOrOptions, debugOptions, isSSR = false) {  
 let getter;  
 let setter;  
 const onlyGetter = isFunction(getterOrOptions);  
 if (onlyGetter) {  
 getter = getterOrOptions;  
 setter = true ? () => {  
 warn("Write operation failed: computed value is readonly");  
 } : NOOP;  
 } else {  
 getter = getterOrOptions.get;  
 setter = getterOrOptions.set;  
 }  
 const cRef = new ComputedRefImpl(getter, setter, onlyGetter || !setter, isSSR);  
 if (debugOptions && !isSSR) {  
 cRef.effect.onTrack = debugOptions.onTrack;  
 cRef.effect.onTrigger = debugOptions.onTrigger;  
 }  
 return cRef;  
}  
function trackRefValue(ref2) {  
 var \_a;  
 if (shouldTrack && activeEffect) {  
 ref2 = toRaw(ref2);  
 trackEffect(  
 activeEffect,  
 (\_a = ref2.dep) != null ? \_a : ref2.dep = createDep(  
 () => ref2.dep = void 0,  
 ref2 instanceof ComputedRefImpl ? ref2 : void 0  
 ),  
 true ? {  
 target: ref2,  
 type: "get",  
 key: "value"  
 } : void 0  
 );  
 }  
}  
function triggerRefValue(ref2, dirtyLevel = 4, newVal, oldVal) {  
 ref2 = toRaw(ref2);  
 const dep = ref2.dep;  
 if (dep) {  
 triggerEffects(  
 dep,  
 dirtyLevel,  
 true ? {  
 target: ref2,  
 type: "set",  
 key: "value",  
 newValue: newVal,  
 oldValue: oldVal  
 } : void 0  
 );  
 }  
}  
function isRef2(r) {  
 return !!(r && r.\_\_v\_isRef === true);  
}  
function ref(value) {  
 return createRef(value, false);  
}  
function shallowRef(value) {  
 return createRef(value, true);  
}  
function createRef(rawValue, shallow) {  
 if (isRef2(rawValue)) {  
 return rawValue;  
 }  
 return new RefImpl(rawValue, shallow);  
}  
var RefImpl = class {  
 constructor(value, \_\_v\_isShallow) {  
 this.\_\_v\_isShallow = \_\_v\_isShallow;  
 this.dep = void 0;  
 this.\_\_v\_isRef = true;  
 this.\_rawValue = \_\_v\_isShallow ? value : toRaw(value);  
 this.\_value = \_\_v\_isShallow ? value : toReactive(value);  
 }  
 get value() {  
 trackRefValue(this);  
 return this.\_value;  
 }  
 set value(newVal) {  
 const useDirectValue = this.\_\_v\_isShallow || isShallow(newVal) || isReadonly(newVal);  
 newVal = useDirectValue ? newVal : toRaw(newVal);  
 if (hasChanged(newVal, this.\_rawValue)) {  
 const oldVal = this.\_rawValue;  
 this.\_rawValue = newVal;  
 this.\_value = useDirectValue ? newVal : toReactive(newVal);  
 triggerRefValue(this, 4, newVal, oldVal);  
 }  
 }  
};  
function triggerRef(ref2) {  
 triggerRefValue(ref2, 4, true ? ref2.value : void 0);  
}  
function unref(ref2) {  
 return isRef2(ref2) ? ref2.value : ref2;  
}  
function toValue(source) {  
 return isFunction(source) ? source() : unref(source);  
}  
var shallowUnwrapHandlers = {  
 get: (target, key, receiver) => unref(Reflect.get(target, key, receiver)),  
 set: (target, key, value, receiver) => {  
 const oldValue = target[key];  
 if (isRef2(oldValue) && !isRef2(value)) {  
 oldValue.value = value;  
 return true;  
 } else {  
 return Reflect.set(target, key, value, receiver);  
 }  
 }  
};  
function proxyRefs(objectWithRefs) {  
 return isReactive(objectWithRefs) ? objectWithRefs : new Proxy(objectWithRefs, shallowUnwrapHandlers);  
}  
var CustomRefImpl = class {  
 constructor(factory) {  
 this.dep = void 0;  
 this.\_\_v\_isRef = true;  
 const { get: get2, set: set2 } = factory(  
 () => trackRefValue(this),  
 () => triggerRefValue(this)  
 );  
 this.\_get = get2;  
 this.\_set = set2;  
 }  
 get value() {  
 return this.\_get();  
 }  
 set value(newVal) {  
 this.\_set(newVal);  
 }  
};  
function customRef(factory) {  
 return new CustomRefImpl(factory);  
}  
function toRefs(object) {  
 if (!isProxy(object)) {  
 warn(`toRefs() expects a reactive object but received a plain one.`);  
 }  
 const ret = isArray(object) ? new Array(object.length) : {};  
 for (const key in object) {  
 ret[key] = propertyToRef(object, key);  
 }  
 return ret;  
}  
var ObjectRefImpl = class {  
 constructor(\_object, \_key, \_defaultValue) {  
 this.\_object = \_object;  
 this.\_key = \_key;  
 this.\_defaultValue = \_defaultValue;  
 this.\_\_v\_isRef = true;  
 }  
 get value() {  
 const val = this.\_object[this.\_key];  
 return val === void 0 ? this.\_defaultValue : val;  
 }  
 set value(newVal) {  
 this.\_object[this.\_key] = newVal;  
 }  
 get dep() {  
 return getDepFromReactive(toRaw(this.\_object), this.\_key);  
 }  
};  
var GetterRefImpl = class {  
 constructor(\_getter) {  
 this.\_getter = \_getter;  
 this.\_\_v\_isRef = true;  
 this.\_\_v\_isReadonly = true;  
 }  
 get value() {  
 return this.\_getter();  
 }  
};  
function toRef(source, key, defaultValue) {  
 if (isRef2(source)) {  
 return source;  
 } else if (isFunction(source)) {  
 return new GetterRefImpl(source);  
 } else if (isObject(source) && arguments.length > 1) {  
 return propertyToRef(source, key, defaultValue);  
 } else {  
 return ref(source);  
 }  
}  
function propertyToRef(source, key, defaultValue) {  
 const val = source[key];  
 return isRef2(val) ? val : new ObjectRefImpl(source, key, defaultValue);  
}  
var TrackOpTypes = {  
 "GET": "get",  
 "HAS": "has",  
 "ITERATE": "iterate"  
};  
var TriggerOpTypes = {  
 "SET": "set",  
 "ADD": "add",  
 "DELETE": "delete",  
 "CLEAR": "clear"  
};  
var stack = [];  
function pushWarningContext(vnode) {  
 stack.push(vnode);  
}  
function popWarningContext() {  
 stack.pop();  
}  
var isWarning = false;  
function warn$1(msg, ...args) {  
 if (isWarning) return;  
 isWarning = true;  
 pauseTracking();  
 const instance = stack.length ? stack[stack.length - 1].component : null;  
 const appWarnHandler = instance && instance.appContext.config.warnHandler;  
 const trace = getComponentTrace();  
 if (appWarnHandler) {  
 callWithErrorHandling(  
 appWarnHandler,  
 instance,  
 11,  
 [  
 msg + args.map((a) => {  
 var \_a, \_b;  
 return (\_b = (\_a = a.toString) == null ? void 0 : \_a.call(a)) != null ? \_b : JSON.stringify(a);  
 }).join(""),  
 instance && instance.proxy,  
 trace.map(  
 ({ vnode }) => `at <${formatComponentName(instance, vnode.type)}>`  
 ).join("\n"),  
 trace  
 ]  
 );  
 } else {  
 const warnArgs = [`[Vue warn]: ${msg}`, ...args];  
 if (trace.length && // avoid spamming console during tests  
 true) {  
 warnArgs.push(`  
`, ...formatTrace(trace));  
 }  
 console.warn(...warnArgs);  
 }  
 resetTracking();  
 isWarning = false;  
}  
function getComponentTrace() {  
 let currentVNode = stack[stack.length - 1];  
 if (!currentVNode) {  
 return [];  
 }  
 const normalizedStack = [];  
 while (currentVNode) {  
 const last = normalizedStack[0];  
 if (last && last.vnode === currentVNode) {  
 last.recurseCount++;  
 } else {  
 normalizedStack.push({  
 vnode: currentVNode,  
 recurseCount: 0  
 });  
 }  
 const parentInstance = currentVNode.component && currentVNode.component.parent;  
 currentVNode = parentInstance && parentInstance.vnode;  
 }  
 return normalizedStack;  
}  
function formatTrace(trace) {  
 const logs = [];  
 trace.forEach((entry, i) => {  
 logs.push(...i === 0 ? [] : [`  
`], ...formatTraceEntry(entry));  
 });  
 return logs;  
}  
function formatTraceEntry({ vnode, recurseCount }) {  
 const postfix = recurseCount > 0 ? `... (${recurseCount} recursive calls)` : ``;  
 const isRoot = vnode.component ? vnode.component.parent == null : false;  
 const open = ` at <${formatComponentName(  
 vnode.component,  
 vnode.type,  
 isRoot  
 )}`;  
 const close = `>` + postfix;  
 return vnode.props ? [open, ...formatProps(vnode.props), close] : [open + close];  
}  
function formatProps(props) {  
 const res = [];  
 const keys = Object.keys(props);  
 keys.slice(0, 3).forEach((key) => {  
 res.push(...formatProp(key, props[key]));  
 });  
 if (keys.length > 3) {  
 res.push(` ...`);  
 }  
 return res;  
}  
function formatProp(key, value, raw) {  
 if (isString(value)) {  
 value = JSON.stringify(value);  
 return raw ? value : [`${key}=${value}`];  
 } else if (typeof value === "number" || typeof value === "boolean" || value == null) {  
 return raw ? value : [`${key}=${value}`];  
 } else if (isRef2(value)) {  
 value = formatProp(key, toRaw(value.value), true);  
 return raw ? value : [`${key}=Ref<`, value, `>`];  
 } else if (isFunction(value)) {  
 return [`${key}=fn${value.name ? `<${value.name}>` : ``}`];  
 } else {  
 value = toRaw(value);  
 return raw ? value : [`${key}=`, value];  
 }  
}  
function assertNumber(val, type) {  
 if (false) return;  
 if (val === void 0) {  
 return;  
 } else if (typeof val !== "number") {  
 warn$1(`${type} is not a valid number - got ${JSON.stringify(val)}.`);  
 } else if (isNaN(val)) {  
 warn$1(`${type} is NaN - the duration expression might be incorrect.`);  
 }  
}  
var ErrorCodes = {  
 "SETUP\_FUNCTION": 0,  
 "0": "SETUP\_FUNCTION",  
 "RENDER\_FUNCTION": 1,  
 "1": "RENDER\_FUNCTION",  
 "WATCH\_GETTER": 2,  
 "2": "WATCH\_GETTER",  
 "WATCH\_CALLBACK": 3,  
 "3": "WATCH\_CALLBACK",  
 "WATCH\_CLEANUP": 4,  
 "4": "WATCH\_CLEANUP",  
 "NATIVE\_EVENT\_HANDLER": 5,  
 "5": "NATIVE\_EVENT\_HANDLER",  
 "COMPONENT\_EVENT\_HANDLER": 6,  
 "6": "COMPONENT\_EVENT\_HANDLER",  
 "VNODE\_HOOK": 7,  
 "7": "VNODE\_HOOK",  
 "DIRECTIVE\_HOOK": 8,  
 "8": "DIRECTIVE\_HOOK",  
 "TRANSITION\_HOOK": 9,  
 "9": "TRANSITION\_HOOK",  
 "APP\_ERROR\_HANDLER": 10,  
 "10": "APP\_ERROR\_HANDLER",  
 "APP\_WARN\_HANDLER": 11,  
 "11": "APP\_WARN\_HANDLER",  
 "FUNCTION\_REF": 12,  
 "12": "FUNCTION\_REF",  
 "ASYNC\_COMPONENT\_LOADER": 13,  
 "13": "ASYNC\_COMPONENT\_LOADER",  
 "SCHEDULER": 14,  
 "14": "SCHEDULER",  
 "COMPONENT\_UPDATE": 15,  
 "15": "COMPONENT\_UPDATE"  
};  
var ErrorTypeStrings$1 = {  
 ["sp"]: "serverPrefetch hook",  
 ["bc"]: "beforeCreate hook",  
 ["c"]: "created hook",  
 ["bm"]: "beforeMount hook",  
 ["m"]: "mounted hook",  
 ["bu"]: "beforeUpdate hook",  
 ["u"]: "updated",  
 ["bum"]: "beforeUnmount hook",  
 ["um"]: "unmounted hook",  
 ["a"]: "activated hook",  
 ["da"]: "deactivated hook",  
 ["ec"]: "errorCaptured hook",  
 ["rtc"]: "renderTracked hook",  
 ["rtg"]: "renderTriggered hook",  
 [0]: "setup function",  
 [1]: "render function",  
 [2]: "watcher getter",  
 [3]: "watcher callback",  
 [4]: "watcher cleanup function",  
 [5]: "native event handler",  
 [6]: "component event handler",  
 [7]: "vnode hook",  
 [8]: "directive hook",  
 [9]: "transition hook",  
 [10]: "app errorHandler",  
 [11]: "app warnHandler",  
 [12]: "ref function",  
 [13]: "async component loader",  
 [14]: "scheduler flush",  
 [15]: "component update"  
};  
function callWithErrorHandling(fn, instance, type, args) {  
 try {  
 return args ? fn(...args) : fn();  
 } catch (err) {  
 handleError(err, instance, type);  
 }  
}  
function callWithAsyncErrorHandling(fn, instance, type, args) {  
 if (isFunction(fn)) {  
 const res = callWithErrorHandling(fn, instance, type, args);  
 if (res && isPromise(res)) {  
 res.catch((err) => {  
 handleError(err, instance, type);  
 });  
 }  
 return res;  
 }  
 if (isArray(fn)) {  
 const values = [];  
 for (let i = 0; i < fn.length; i++) {  
 values.push(callWithAsyncErrorHandling(fn[i], instance, type, args));  
 }  
 return values;  
 } else if (true) {  
 warn$1(  
 `Invalid value type passed to callWithAsyncErrorHandling(): ${typeof fn}`  
 );  
 }  
}  
function handleError(err, instance, type, throwInDev = true) {  
 const contextVNode = instance ? instance.vnode : null;  
 if (instance) {  
 let cur = instance.parent;  
 const exposedInstance = instance.proxy;  
 const errorInfo = true ? ErrorTypeStrings$1[type] : `https://vuejs.org/error-reference/#runtime-${type}`;  
 while (cur) {  
 const errorCapturedHooks = cur.ec;  
 if (errorCapturedHooks) {  
 for (let i = 0; i < errorCapturedHooks.length; i++) {  
 if (errorCapturedHooks[i](err, exposedInstance, errorInfo) === false) {  
 return;  
 }  
 }  
 }  
 cur = cur.parent;  
 }  
 const appErrorHandler = instance.appContext.config.errorHandler;  
 if (appErrorHandler) {  
 pauseTracking();  
 callWithErrorHandling(  
 appErrorHandler,  
 null,  
 10,  
 [err, exposedInstance, errorInfo]  
 );  
 resetTracking();  
 return;  
 }  
 }  
 logError(err, type, contextVNode, throwInDev);  
}  
function logError(err, type, contextVNode, throwInDev = true) {  
 if (true) {  
 const info = ErrorTypeStrings$1[type];  
 if (contextVNode) {  
 pushWarningContext(contextVNode);  
 }  
 warn$1(`Unhandled error${info ? ` during execution of ${info}` : ``}`);  
 if (contextVNode) {  
 popWarningContext();  
 }  
 if (throwInDev) {  
 throw err;  
 } else {  
 console.error(err);  
 }  
 } else {  
 console.error(err);  
 }  
}  
var isFlushing = false;  
var isFlushPending = false;  
var queue = [];  
var flushIndex = 0;  
var pendingPostFlushCbs = [];  
var activePostFlushCbs = null;  
var postFlushIndex = 0;  
var resolvedPromise = Promise.resolve();  
var currentFlushPromise = null;  
var RECURSION\_LIMIT = 100;  
function nextTick(fn) {  
 const p2 = currentFlushPromise || resolvedPromise;  
 return fn ? p2.then(this ? fn.bind(this) : fn) : p2;  
}  
function findInsertionIndex(id) {  
 let start = flushIndex + 1;  
 let end = queue.length;  
 while (start < end) {  
 const middle = start + end >>> 1;  
 const middleJob = queue[middle];  
 const middleJobId = getId(middleJob);  
 if (middleJobId < id || middleJobId === id && middleJob.pre) {  
 start = middle + 1;  
 } else {  
 end = middle;  
 }  
 }  
 return start;  
}  
function queueJob(job) {  
 if (!queue.length || !queue.includes(  
 job,  
 isFlushing && job.allowRecurse ? flushIndex + 1 : flushIndex  
 )) {  
 if (job.id == null) {  
 queue.push(job);  
 } else {  
 queue.splice(findInsertionIndex(job.id), 0, job);  
 }  
 queueFlush();  
 }  
}  
function queueFlush() {  
 if (!isFlushing && !isFlushPending) {  
 isFlushPending = true;  
 currentFlushPromise = resolvedPromise.then(flushJobs);  
 }  
}  
function invalidateJob(job) {  
 const i = queue.indexOf(job);  
 if (i > flushIndex) {  
 queue.splice(i, 1);  
 }  
}  
function queuePostFlushCb(cb) {  
 if (!isArray(cb)) {  
 if (!activePostFlushCbs || !activePostFlushCbs.includes(  
 cb,  
 cb.allowRecurse ? postFlushIndex + 1 : postFlushIndex  
 )) {  
 pendingPostFlushCbs.push(cb);  
 }  
 } else {  
 pendingPostFlushCbs.push(...cb);  
 }  
 queueFlush();  
}  
function flushPreFlushCbs(instance, seen, i = isFlushing ? flushIndex + 1 : 0) {  
 if (true) {  
 seen = seen || /\* @\_\_PURE\_\_ \*/ new Map();  
 }  
 for (; i < queue.length; i++) {  
 const cb = queue[i];  
 if (cb && cb.pre) {  
 if (instance && cb.id !== instance.uid) {  
 continue;  
 }  
 if (checkRecursiveUpdates(seen, cb)) {  
 continue;  
 }  
 queue.splice(i, 1);  
 i--;  
 cb();  
 }  
 }  
}  
function flushPostFlushCbs(seen) {  
 if (pendingPostFlushCbs.length) {  
 const deduped = [...new Set(pendingPostFlushCbs)].sort(  
 (a, b) => getId(a) - getId(b)  
 );  
 pendingPostFlushCbs.length = 0;  
 if (activePostFlushCbs) {  
 activePostFlushCbs.push(...deduped);  
 return;  
 }  
 activePostFlushCbs = deduped;  
 if (true) {  
 seen = seen || /\* @\_\_PURE\_\_ \*/ new Map();  
 }  
 for (postFlushIndex = 0; postFlushIndex < activePostFlushCbs.length; postFlushIndex++) {  
 const cb = activePostFlushCbs[postFlushIndex];  
 if (checkRecursiveUpdates(seen, cb)) {  
 continue;  
 }  
 if (cb.active !== false) cb();  
 }  
 activePostFlushCbs = null;  
 postFlushIndex = 0;  
 }  
}  
var getId = (job) => job.id == null ? Infinity : job.id;  
var comparator = (a, b) => {  
 const diff = getId(a) - getId(b);  
 if (diff === 0) {  
 if (a.pre && !b.pre) return -1;  
 if (b.pre && !a.pre) return 1;  
 }  
 return diff;  
};  
function flushJobs(seen) {  
 isFlushPending = false;  
 isFlushing = true;  
 if (true) {  
 seen = seen || /\* @\_\_PURE\_\_ \*/ new Map();  
 }  
 queue.sort(comparator);  
 const check = true ? (job) => checkRecursiveUpdates(seen, job) : NOOP;  
 try {  
 for (flushIndex = 0; flushIndex < queue.length; flushIndex++) {  
 const job = queue[flushIndex];  
 if (job && job.active !== false) {  
 if (check(job)) {  
 continue;  
 }  
 callWithErrorHandling(  
 job,  
 job.i,  
 job.i ? 15 : 14  
 );  
 }  
 }  
 } finally {  
 flushIndex = 0;  
 queue.length = 0;  
 flushPostFlushCbs(seen);  
 isFlushing = false;  
 currentFlushPromise = null;  
 if (queue.length || pendingPostFlushCbs.length) {  
 flushJobs(seen);  
 }  
 }  
}  
function checkRecursiveUpdates(seen, fn) {  
 if (!seen.has(fn)) {  
 seen.set(fn, 1);  
 } else {  
 const count = seen.get(fn);  
 if (count > RECURSION\_LIMIT) {  
 const instance = fn.i;  
 const componentName = instance && getComponentName(instance.type);  
 handleError(  
 `Maximum recursive updates exceeded${componentName ? ` in component <${componentName}>` : ``}. This means you have a reactive effect that is mutating its own dependencies and thus recursively triggering itself. Possible sources include component template, render function, updated hook or watcher source function.`,  
 null,  
 10  
 );  
 return true;  
 } else {  
 seen.set(fn, count + 1);  
 }  
 }  
}  
var isHmrUpdating = false;  
var hmrDirtyComponents = /\* @\_\_PURE\_\_ \*/ new Map();  
if (true) {  
 getGlobalThis().\_\_VUE\_HMR\_RUNTIME\_\_ = {  
 createRecord: tryWrap(createRecord),  
 rerender: tryWrap(rerender),  
 reload: tryWrap(reload)  
 };  
}  
var map = /\* @\_\_PURE\_\_ \*/ new Map();  
function registerHMR(instance) {  
 const id = instance.type.\_\_hmrId;  
 let record = map.get(id);  
 if (!record) {  
 createRecord(id, instance.type);  
 record = map.get(id);  
 }  
 record.instances.add(instance);  
}  
function unregisterHMR(instance) {  
 map.get(instance.type.\_\_hmrId).instances.delete(instance);  
}  
function createRecord(id, initialDef) {  
 if (map.has(id)) {  
 return false;  
 }  
 map.set(id, {  
 initialDef: normalizeClassComponent(initialDef),  
 instances: /\* @\_\_PURE\_\_ \*/ new Set()  
 });  
 return true;  
}  
function normalizeClassComponent(component) {  
 return isClassComponent(component) ? component.\_\_vccOpts : component;  
}  
function rerender(id, newRender) {  
 const record = map.get(id);  
 if (!record) {  
 return;  
 }  
 record.initialDef.render = newRender;  
 [...record.instances].forEach((instance) => {  
 if (newRender) {  
 instance.render = newRender;  
 normalizeClassComponent(instance.type).render = newRender;  
 }  
 instance.renderCache = [];  
 isHmrUpdating = true;  
 instance.effect.dirty = true;  
 instance.update();  
 isHmrUpdating = false;  
 });  
}  
function reload(id, newComp) {  
 const record = map.get(id);  
 if (!record) return;  
 newComp = normalizeClassComponent(newComp);  
 updateComponentDef(record.initialDef, newComp);  
 const instances = [...record.instances];  
 for (let i = 0; i < instances.length; i++) {  
 const instance = instances[i];  
 const oldComp = normalizeClassComponent(instance.type);  
 let dirtyInstances = hmrDirtyComponents.get(oldComp);  
 if (!dirtyInstances) {  
 if (oldComp !== record.initialDef) {  
 updateComponentDef(oldComp, newComp);  
 }  
 hmrDirtyComponents.set(oldComp, dirtyInstances = /\* @\_\_PURE\_\_ \*/ new Set());  
 }  
 dirtyInstances.add(instance);  
 instance.appContext.propsCache.delete(instance.type);  
 instance.appContext.emitsCache.delete(instance.type);  
 instance.appContext.optionsCache.delete(instance.type);  
 if (instance.ceReload) {  
 dirtyInstances.add(instance);  
 instance.ceReload(newComp.styles);  
 dirtyInstances.delete(instance);  
 } else if (instance.parent) {  
 instance.parent.effect.dirty = true;  
 queueJob(() => {  
 instance.parent.update();  
 dirtyInstances.delete(instance);  
 });  
 } else if (instance.appContext.reload) {  
 instance.appContext.reload();  
 } else if (typeof window !== "undefined") {  
 window.location.reload();  
 } else {  
 console.warn(  
 "[HMR] Root or manually mounted instance modified. Full reload required."  
 );  
 }  
 }  
 queuePostFlushCb(() => {  
 hmrDirtyComponents.clear();  
 });  
}  
function updateComponentDef(oldComp, newComp) {  
 extend(oldComp, newComp);  
 for (const key in oldComp) {  
 if (key !== "\_\_file" && !(key in newComp)) {  
 delete oldComp[key];  
 }  
 }  
}  
function tryWrap(fn) {  
 return (id, arg) => {  
 try {  
 return fn(id, arg);  
 } catch (e) {  
 console.error(e);  
 console.warn(  
 `[HMR] Something went wrong during Vue component hot-reload. Full reload required.`  
 );  
 }  
 };  
}  
var devtools$1;  
var buffer = [];  
var devtoolsNotInstalled = false;  
function emit$1(event, ...args) {  
 if (devtools$1) {  
 devtools$1.emit(event, ...args);  
 } else if (!devtoolsNotInstalled) {  
 buffer.push({ event, args });  
 }  
}  
function setDevtoolsHook$1(hook, target) {  
 var \_a, \_b;  
 devtools$1 = hook;  
 if (devtools$1) {  
 devtools$1.enabled = true;  
 buffer.forEach(({ event, args }) => devtools$1.emit(event, ...args));  
 buffer = [];  
 } else if (  
 typeof window !== "undefined" && // some envs mock window but not fully  
 window.HTMLElement && // also exclude jsdom  
 !((\_b = (\_a = window.navigator) == null ? void 0 : \_a.userAgent) == null ? void 0 : \_b.includes("jsdom"))  
 ) {  
 const replay = target.\_\_VUE\_DEVTOOLS\_HOOK\_REPLAY\_\_ = target.\_\_VUE\_DEVTOOLS\_HOOK\_REPLAY\_\_ || [];  
 replay.push((newHook) => {  
 setDevtoolsHook$1(newHook, target);  
 });  
 setTimeout(() => {  
 if (!devtools$1) {  
 target.\_\_VUE\_DEVTOOLS\_HOOK\_REPLAY\_\_ = null;  
 devtoolsNotInstalled = true;  
 buffer = [];  
 }  
 }, 3e3);  
 } else {  
 devtoolsNotInstalled = true;  
 buffer = [];  
 }  
}  
function devtoolsInitApp(app, version2) {  
 emit$1("app:init", app, version2, {  
 Fragment,  
 Text,  
 Comment,  
 Static  
 });  
}  
function devtoolsUnmountApp(app) {  
 emit$1("app:unmount", app);  
}  
var devtoolsComponentAdded = createDevtoolsComponentHook(  
 "component:added"  
);  
var devtoolsComponentUpdated = createDevtoolsComponentHook(  
 "component:updated"  
);  
var \_devtoolsComponentRemoved = createDevtoolsComponentHook(  
 "component:removed"  
);  
var devtoolsComponentRemoved = (component) => {  
 if (devtools$1 && typeof devtools$1.cleanupBuffer === "function" && // remove the component if it wasn't buffered  
 !devtools$1.cleanupBuffer(component)) {  
 \_devtoolsComponentRemoved(component);  
 }  
};  
function createDevtoolsComponentHook(hook) {  
 return (component) => {  
 emit$1(  
 hook,  
 component.appContext.app,  
 component.uid,  
 component.parent ? component.parent.uid : void 0,  
 component  
 );  
 };  
}  
var devtoolsPerfStart = createDevtoolsPerformanceHook(  
 "perf:start"  
);  
var devtoolsPerfEnd = createDevtoolsPerformanceHook(  
 "perf:end"  
);  
function createDevtoolsPerformanceHook(hook) {  
 return (component, type, time) => {  
 emit$1(hook, component.appContext.app, component.uid, component, type, time);  
 };  
}  
function devtoolsComponentEmit(component, event, params) {  
 emit$1(  
 "component:emit",  
 component.appContext.app,  
 component,  
 event,  
 params  
 );  
}  
var currentRenderingInstance = null;  
var currentScopeId = null;  
function setCurrentRenderingInstance(instance) {  
 const prev = currentRenderingInstance;  
 currentRenderingInstance = instance;  
 currentScopeId = instance && instance.type.\_\_scopeId || null;  
 return prev;  
}  
function pushScopeId(id) {  
 currentScopeId = id;  
}  
function popScopeId() {  
 currentScopeId = null;  
}  
var withScopeId = (\_id) => withCtx;  
function withCtx(fn, ctx = currentRenderingInstance, isNonScopedSlot) {  
 if (!ctx) return fn;  
 if (fn.\_n) {  
 return fn;  
 }  
 const renderFnWithContext = (...args) => {  
 if (renderFnWithContext.\_d) {  
 setBlockTracking(-1);  
 }  
 const prevInstance = setCurrentRenderingInstance(ctx);  
 let res;  
 try {  
 res = fn(...args);  
 } finally {  
 setCurrentRenderingInstance(prevInstance);  
 if (renderFnWithContext.\_d) {  
 setBlockTracking(1);  
 }  
 }  
 if (true) {  
 devtoolsComponentUpdated(ctx);  
 }  
 return res;  
 };  
 renderFnWithContext.\_n = true;  
 renderFnWithContext.\_c = true;  
 renderFnWithContext.\_d = true;  
 return renderFnWithContext;  
}  
function validateDirectiveName(name) {  
 if (isBuiltInDirective(name)) {  
 warn$1("Do not use built-in directive ids as custom directive id: " + name);  
 }  
}  
function withDirectives(vnode, directives) {  
 if (currentRenderingInstance === null) {  
 warn$1(`withDirectives can only be used inside render functions.`);  
 return vnode;  
 }  
 const instance = getComponentPublicInstance(currentRenderingInstance);  
 const bindings = vnode.dirs || (vnode.dirs = []);  
 for (let i = 0; i < directives.length; i++) {  
 let [dir, value, arg, modifiers = EMPTY\_OBJ] = directives[i];  
 if (dir) {  
 if (isFunction(dir)) {  
 dir = {  
 mounted: dir,  
 updated: dir  
 };  
 }  
 if (dir.deep) {  
 traverse(value);  
 }  
 bindings.push({  
 dir,  
 instance,  
 value,  
 oldValue: void 0,  
 arg,  
 modifiers  
 });  
 }  
 }  
 return vnode;  
}  
function invokeDirectiveHook(vnode, prevVNode, instance, name) {  
 const bindings = vnode.dirs;  
 const oldBindings = prevVNode && prevVNode.dirs;  
 for (let i = 0; i < bindings.length; i++) {  
 const binding = bindings[i];  
 if (oldBindings) {  
 binding.oldValue = oldBindings[i].value;  
 }  
 let hook = binding.dir[name];  
 if (hook) {  
 pauseTracking();  
 callWithAsyncErrorHandling(hook, instance, 8, [  
 vnode.el,  
 binding,  
 vnode,  
 prevVNode  
 ]);  
 resetTracking();  
 }  
 }  
}  
var leaveCbKey = Symbol("\_leaveCb");  
var enterCbKey = Symbol("\_enterCb");  
function useTransitionState() {  
 const state = {  
 isMounted: false,  
 isLeaving: false,  
 isUnmounting: false,  
 leavingVNodes: /\* @\_\_PURE\_\_ \*/ new Map()  
 };  
 onMounted(() => {  
 state.isMounted = true;  
 });  
 onBeforeUnmount(() => {  
 state.isUnmounting = true;  
 });  
 return state;  
}  
var TransitionHookValidator = [Function, Array];  
var BaseTransitionPropsValidators = {  
 mode: String,  
 appear: Boolean,  
 persisted: Boolean,  
 onBeforeEnter: TransitionHookValidator,  
 onEnter: TransitionHookValidator,  
 onAfterEnter: TransitionHookValidator,  
 onEnterCancelled: TransitionHookValidator,  
 onBeforeLeave: TransitionHookValidator,  
 onLeave: TransitionHookValidator,  
 onAfterLeave: TransitionHookValidator,  
 onLeaveCancelled: TransitionHookValidator,  
 onBeforeAppear: TransitionHookValidator,  
 onAppear: TransitionHookValidator,  
 onAfterAppear: TransitionHookValidator,  
 onAppearCancelled: TransitionHookValidator  
};  
var recursiveGetSubtree = (instance) => {  
 const subTree = instance.subTree;  
 return subTree.component ? recursiveGetSubtree(subTree.component) : subTree;  
};  
var BaseTransitionImpl = {  
 name: `BaseTransition`,  
 props: BaseTransitionPropsValidators,  
 setup(props, { slots }) {  
 const instance = getCurrentInstance();  
 const state = useTransitionState();  
 return () => {  
 const children = slots.default && getTransitionRawChildren(slots.default(), true);  
 if (!children || !children.length) {  
 return;  
 }  
 let child = children[0];  
 if (children.length > 1) {  
 let hasFound = false;  
 for (const c of children) {  
 if (c.type !== Comment) {  
 if (hasFound) {  
 warn$1(  
 "<transition> can only be used on a single element or component. Use <transition-group> for lists."  
 );  
 break;  
 }  
 child = c;  
 hasFound = true;  
 if (false) break;  
 }  
 }  
 }  
 const rawProps = toRaw(props);  
 const { mode } = rawProps;  
  
 warn$1(`invalid <transition> mode: ${mode}`);  
 }  
 if (state.isLeaving) {  
 return emptyPlaceholder(child);  
 }  
 const innerChild = getKeepAliveChild(child);  
 if (!innerChild) {  
 return emptyPlaceholder(child);  
 }  
 let enterHooks = resolveTransitionHooks(  
 innerChild,  
 rawProps,  
 state,  
 instance,  
 (hooks) => enterHooks = hooks  
 );  
 setTransitionHooks(innerChild, enterHooks);  
 const oldChild = instance.subTree;  
 const oldInnerChild = oldChild && getKeepAliveChild(oldChild);  
  
 const leavingHooks = resolveTransitionHooks(  
 oldInnerChild,  
 rawProps,  
 state,  
 instance  
 );  
 setTransitionHooks(oldInnerChild, leavingHooks);  
 if (mode === "out-in" && innerChild.type !== Comment) {  
 state.isLeaving = true;  
 leavingHooks.afterLeave = () => {  
 state.isLeaving = false;  
 if (instance.update.active !== false) {  
 instance.effect.dirty = true;  
 instance.update();  
 }  
 };  
 return emptyPlaceholder(child);  
 } else if (mode === "in-out" && innerChild.type !== Comment) {  
 leavingHooks.delayLeave = (el, earlyRemove, delayedLeave) => {  
 const leavingVNodesCache = getLeavingNodesForType(  
 state,  
 oldInnerChild  
 );  
 leavingVNodesCache[String(oldInnerChild.key)] = oldInnerChild;  
 el[leaveCbKey] = () => {  
 earlyRemove();  
 el[leaveCbKey] = void 0;  
 delete enterHooks.delayedLeave;  
 };  
 enterHooks.delayedLeave = delayedLeave;  
 };  
 }  
 }  
 return child;  
 };  
 }  
};  
var BaseTransition = BaseTransitionImpl;  
function getLeavingNodesForType(state, vnode) {  
 const { leavingVNodes } = state;  
 let leavingVNodesCache = leavingVNodes.get(vnode.type);  
 if (!leavingVNodesCache) {  
 leavingVNodesCache = /\* @\_\_PURE\_\_ \*/ Object.create(null);  
 leavingVNodes.set(vnode.type, leavingVNodesCache);  
 }  
 return leavingVNodesCache;  
}  
function resolveTransitionHooks(vnode, props, state, instance, postClone) {  
 const {  
 appear,  
 mode,  
 persisted = false,  
 onBeforeEnter,  
 onEnter,  
 onAfterEnter,  
 onEnterCancelled,  
 onBeforeLeave,  
 onLeave,  
 onAfterLeave,  
 onLeaveCancelled,  
 onBeforeAppear,  
 onAppear,  
 onAfterAppear,  
 onAppearCancelled  
 } = props;  
 const key = String(vnode.key);  
 const leavingVNodesCache = getLeavingNodesForType(state, vnode);  
 const callHook3 = (hook, args) => {  
 hook && callWithAsyncErrorHandling(  
 hook,  
 instance,  
 9,  
 args  
 );  
 };  
 const callAsyncHook = (hook, args) => {  
 const done = args[1];  
 callHook3(hook, args);  
 if (isArray(hook)) {  
 if (hook.every((hook2) => hook2.length <= 1)) done();  
 } else if (hook.length <= 1) {  
 done();  
 }  
 };  
 const hooks = {  
 mode,  
 persisted,  
 beforeEnter(el) {  
 let hook = onBeforeEnter;  
 if (!state.isMounted) {  
 if (appear) {  
 hook = onBeforeAppear || onBeforeEnter;  
 } else {  
 return;  
 }  
 }  
 if (el[leaveCbKey]) {  
 el[leaveCbKey](  
 true  
 );  
 }  
 const leavingVNode = leavingVNodesCache[key];  
 if (leavingVNode && isSameVNodeType(vnode, leavingVNode) && leavingVNode.el[leaveCbKey]) {  
 leavingVNode.el[leaveCbKey]();  
 }  
 callHook3(hook, [el]);  
 },  
 enter(el) {  
 let hook = onEnter;  
 let afterHook = onAfterEnter;  
 let cancelHook = onEnterCancelled;  
 if (!state.isMounted) {  
 if (appear) {  
 hook = onAppear || onEnter;  
 afterHook = onAfterAppear || onAfterEnter;  
 cancelHook = onAppearCancelled || onEnterCancelled;  
 } else {  
 return;  
 }  
 }  
 let called = false;  
 const done = el[enterCbKey] = (cancelled) => {  
 if (called) return;  
 called = true;  
 if (cancelled) {  
 callHook3(cancelHook, [el]);  
 } else {  
 callHook3(afterHook, [el]);  
 }  
 if (hooks.delayedLeave) {  
 hooks.delayedLeave();  
 }  
 el[enterCbKey] = void 0;  
 };  
 if (hook) {  
 callAsyncHook(hook, [el, done]);  
 } else {  
 done();  
 }  
 },  
 leave(el, remove2) {  
 const key2 = String(vnode.key);  
 if (el[enterCbKey]) {  
 el[enterCbKey](  
 true  
 );  
 }  
 if (state.isUnmounting) {  
 return remove2();  
 }  
 callHook3(onBeforeLeave, [el]);  
 let called = false;  
 const done = el[leaveCbKey] = (cancelled) => {  
 if (called) return;  
 called = true;  
 remove2();  
 if (cancelled) {  
 callHook3(onLeaveCancelled, [el]);  
 } else {  
 callHook3(onAfterLeave, [el]);  
 }  
 el[leaveCbKey] = void 0;  
 if (leavingVNodesCache[key2] === vnode) {  
 delete leavingVNodesCache[key2];  
 }  
 };  
 leavingVNodesCache[key2] = vnode;  
 if (onLeave) {  
 callAsyncHook(onLeave, [el, done]);  
 } else {  
 done();  
 }  
 },  
 clone(vnode2) {  
 const hooks2 = resolveTransitionHooks(  
 vnode2,  
 props,  
 state,  
 instance,  
 postClone  
 );  
 if (postClone) postClone(hooks2);  
 return hooks2;  
 }  
 };  
 return hooks;  
}  
function emptyPlaceholder(vnode) {  
 if (isKeepAlive(vnode)) {  
 vnode = cloneVNode(vnode);  
 vnode.children = null;  
 return vnode;  
 }  
}  
function getKeepAliveChild(vnode) {  
 if (!isKeepAlive(vnode)) {  
 return vnode;  
 }  
 if (vnode.component) {  
 return vnode.component.subTree;  
 }  
 const { shapeFlag, children } = vnode;  
 if (children) {  
 if (shapeFlag & 16) {  
 return children[0];  
 }  
 if (shapeFlag & 32 && isFunction(children.default)) {  
 return children.default();  
 }  
 }  
}  
function setTransitionHooks(vnode, hooks) {  
 if (vnode.shapeFlag & 6 && vnode.component) {  
 setTransitionHooks(vnode.component.subTree, hooks);  
 } else if (vnode.shapeFlag & 128) {  
 vnode.ssContent.transition = hooks.clone(vnode.ssContent);  
 vnode.ssFallback.transition = hooks.clone(vnode.ssFallback);  
 } else {  
 vnode.transition = hooks;  
 }  
}  
function getTransitionRawChildren(children, keepComment = false, parentKey) {  
 let ret = [];  
 let keyedFragmentCount = 0;  
 for (let i = 0; i < children.length; i++) {  
 let child = children[i];  
 const key = parentKey == null ? child.key : String(parentKey) + String(child.key != null ? child.key : i);  
 if (child.type === Fragment) {  
 if (child.patchFlag & 128) keyedFragmentCount++;  
 ret = ret.concat(  
 getTransitionRawChildren(child.children, keepComment, key)  
 );  
 } else if (keepComment || child.type !== Comment) {  
 ret.push(key != null ? cloneVNode(child, { key }) : child);  
 }  
 }  
 if (keyedFragmentCount > 1) {  
 for (let i = 0; i < ret.length; i++) {  
 ret[i].patchFlag = -2;  
 }  
 }  
 return ret;  
}  
function defineComponent(options, extraOptions) {  
 return isFunction(options) ? (  
 (() => extend({ name: options.name }, extraOptions, { setup: options }))()  
 ) : options;  
}  
var isAsyncWrapper = (i) => !!i.type.\_\_asyncLoader;  
function defineAsyncComponent(source) {  
 if (isFunction(source)) {  
 source = { loader: source };  
 }  
 const {  
 loader,  
 loadingComponent,  
 errorComponent,  
 delay = 200,  
 timeout,  
 suspensible = true,  
 onError: userOnError  
 } = source;  
 let pendingRequest = null;  
 let resolvedComp;  
 let retries = 0;  
 const retry = () => {  
 retries++;  
 pendingRequest = null;  
 return load();  
 };  
 const load = () => {  
 let thisRequest;  
 return pendingRequest || (thisRequest = pendingRequest = loader().catch((err) => {  
 err = err instanceof Error ? err : new Error(String(err));  
 if (userOnError) {  
 return new Promise((resolve2, reject) => {  
 const userRetry = () => resolve2(retry());  
 const userFail = () => reject(err);  
 userOnError(err, userRetry, userFail, retries + 1);  
 });  
 } else {  
 throw err;  
 }  
 }).then((comp) => {  
 if (thisRequest !== pendingRequest && pendingRequest) {  
 return pendingRequest;  
 }  
 if (!comp) {  
 warn$1(  
 `Async component loader resolved to undefined. If you are using retry(), make sure to return its return value.`  
 );  
 }  
 if (comp && (comp.\_\_esModule || comp[Symbol.toStringTag] === "Module")) {  
 comp = comp.default;  
 }  
 if (comp && !isObject(comp) && !isFunction(comp)) {  
 throw new Error(`Invalid async component load result: ${comp}`);  
 }  
 resolvedComp = comp;  
 return comp;  
 }));  
 };  
 return defineComponent({  
 name: "AsyncComponentWrapper",  
 \_\_asyncLoader: load,  
 get \_\_asyncResolved() {  
 return resolvedComp;  
 },  
 setup() {  
 const instance = currentInstance;  
 if (resolvedComp) {  
 return () => createInnerComp(resolvedComp, instance);  
 }  
 const onError = (err) => {  
 pendingRequest = null;  
 handleError(  
 err,  
 instance,  
 13,  
 !errorComponent  
 );  
 };  
 if (suspensible && instance.suspense || isInSSRComponentSetup) {  
 return load().then((comp) => {  
 return () => createInnerComp(comp, instance);  
 }).catch((err) => {  
 onError(err);  
 return () => errorComponent ? createVNode(errorComponent, {  
 error: err  
 }) : null;  
 });  
 }  
 const loaded = ref(false);  
 const error = ref();  
 const delayed = ref(!!delay);  
 if (delay) {  
 setTimeout(() => {  
 delayed.value = false;  
 }, delay);  
 }  
 if (timeout != null) {  
 setTimeout(() => {  
 if (!loaded.value && !error.value) {  
 const err = new Error(  
 `Async component timed out after ${timeout}ms.`  
 );  
 onError(err);  
 error.value = err;  
 }  
 }, timeout);  
 }  
 load().then(() => {  
 loaded.value = true;  
 if (instance.parent && isKeepAlive(instance.parent.vnode)) {  
 instance.parent.effect.dirty = true;  
 queueJob(instance.parent.update);  
 }  
 }).catch((err) => {  
 onError(err);  
 error.value = err;  
 });  
 return () => {  
 if (loaded.value && resolvedComp) {  
 return createInnerComp(resolvedComp, instance);  
 } else if (error.value && errorComponent) {  
 return createVNode(errorComponent, {  
 error: error.value  
 });  
 } else if (loadingComponent && !delayed.value) {  
 return createVNode(loadingComponent);  
 }  
 };  
 }  
 });  
}  
function createInnerComp(comp, parent) {  
 const { ref: ref2, props, children, ce } = parent.vnode;  
 const vnode = createVNode(comp, props, children);  
 vnode.ref = ref2;  
 vnode.ce = ce;  
 delete parent.vnode.ce;  
 return vnode;  
}  
var isKeepAlive = (vnode) => vnode.type.\_\_isKeepAlive;  
var KeepAliveImpl = {  
 name: `KeepAlive`,  
 \_\_isKeepAlive: true,  
 props: {  
 include: [String, RegExp, Array],  
 exclude: [String, RegExp, Array],  
 max: [String, Number]  
 },  
 setup(props, { slots }) {  
 const instance = getCurrentInstance();  
 const sharedContext = instance.ctx;  
 if (!sharedContext.renderer) {  
 return () => {  
 const children = slots.default && slots.default();  
 return children && children.length === 1 ? children[0] : children;  
 };  
 }  
 const cache = /\* @\_\_PURE\_\_ \*/ new Map();  
 const keys = /\* @\_\_PURE\_\_ \*/ new Set();  
 let current = null;  
 if (true) {  
 instance.\_\_v\_cache = cache;  
 }  
 const parentSuspense = instance.suspense;  
 const {  
 renderer: {  
 p: patch,  
 m: move,  
 um: \_unmount,  
 o: { createElement }  
 }  
 } = sharedContext;  
 const storageContainer = createElement("div");  
 sharedContext.activate = (vnode, container, anchor, namespace, optimized) => {  
 const instance2 = vnode.component;  
 move(vnode, container, anchor, 0, parentSuspense);  
 patch(  
 instance2.vnode,  
 vnode,  
 container,  
 anchor,  
 instance2,  
 parentSuspense,  
 namespace,  
 vnode.slotScopeIds,  
 optimized  
 );  
 queuePostRenderEffect(() => {  
 instance2.isDeactivated = false;  
 if (instance2.a) {  
 invokeArrayFns(instance2.a);  
 }  
 const vnodeHook = vnode.props && vnode.props.onVnodeMounted;  
 if (vnodeHook) {  
 invokeVNodeHook(vnodeHook, instance2.parent, vnode);  
 }  
 }, parentSuspense);  
 if (true) {  
 devtoolsComponentAdded(instance2);  
 }  
 };  
 sharedContext.deactivate = (vnode) => {  
 const instance2 = vnode.component;  
 invalidateMount(instance2.m);  
 invalidateMount(instance2.a);  
 move(vnode, storageContainer, null, 1, parentSuspense);  
 queuePostRenderEffect(() => {  
 if (instance2.da) {  
 invokeArrayFns(instance2.da);  
 }  
 const vnodeHook = vnode.props && vnode.props.onVnodeUnmounted;  
 if (vnodeHook) {  
 invokeVNodeHook(vnodeHook, instance2.parent, vnode);  
 }  
 instance2.isDeactivated = true;  
 }, parentSuspense);  
 if (true) {  
 devtoolsComponentAdded(instance2);  
 }  
 };  
 function unmount(vnode) {  
 resetShapeFlag(vnode);  
 \_unmount(vnode, instance, parentSuspense, true);  
 }  
 function pruneCache(filter) {  
 cache.forEach((vnode, key) => {  
 const name = getComponentName(vnode.type);  
 if (name && (!filter || !filter(name))) {  
 pruneCacheEntry(key);  
 }  
 });  
 }  
 function pruneCacheEntry(key) {  
 const cached = cache.get(key);  
 if (!current || !isSameVNodeType(cached, current)) {  
 unmount(cached);  
 } else if (current) {  
 resetShapeFlag(current);  
 }  
 cache.delete(key);  
 keys.delete(key);  
 }  
 watch(  
 () => [props.include, props.exclude],  
 ([include, exclude]) => {  
 include && pruneCache((name) => matches(include, name));  
 exclude && pruneCache((name) => !matches(exclude, name));  
 },  
 { flush: "post", deep: true }  
 );  
 let pendingCacheKey = null;  
 const cacheSubtree = () => {  
 if (pendingCacheKey != null) {  
 if (isSuspense(instance.subTree.type)) {  
 queuePostRenderEffect(() => {  
 cache.set(pendingCacheKey, getInnerChild(instance.subTree));  
 }, instance.subTree.suspense);  
 } else {  
 cache.set(pendingCacheKey, getInnerChild(instance.subTree));  
 }  
 }  
 };  
 onMounted(cacheSubtree);  
 onUpdated(cacheSubtree);  
 onBeforeUnmount(() => {  
 cache.forEach((cached) => {  
 const { subTree, suspense } = instance;  
 const vnode = getInnerChild(subTree);  
 if (cached.type === vnode.type && cached.key === vnode.key) {  
 resetShapeFlag(vnode);  
 const da = vnode.component.da;  
 da && queuePostRenderEffect(da, suspense);  
 return;  
 }  
 unmount(cached);  
 });  
 });  
 return () => {  
 pendingCacheKey = null;  
 if (!slots.default) {  
 return null;  
 }  
 const children = slots.default();  
 const rawVNode = children[0];  
 if (children.length > 1) {  
 if (true) {  
 warn$1(`KeepAlive should contain exactly one component child.`);  
 }  
 current = null;  
 return children;  
 } else if (!isVNode(rawVNode) || !(rawVNode.shapeFlag & 4) && !(rawVNode.shapeFlag & 128)) {  
 current = null;  
 return rawVNode;  
 }  
 let vnode = getInnerChild(rawVNode);  
 const comp = vnode.type;  
 const name = getComponentName(  
 isAsyncWrapper(vnode) ? vnode.type.\_\_asyncResolved || {} : comp  
 );  
 const { include, exclude, max } = props;  
  
 current = vnode;  
 return rawVNode;  
 }  
 const key = vnode.key == null ? comp : vnode.key;  
 const cachedVNode = cache.get(key);  
 if (vnode.el) {  
 vnode = cloneVNode(vnode);  
 if (rawVNode.shapeFlag & 128) {  
 rawVNode.ssContent = vnode;  
 }  
 }  
 pendingCacheKey = key;  
 if (cachedVNode) {  
 vnode.el = cachedVNode.el;  
 vnode.component = cachedVNode.component;  
 if (vnode.transition) {  
 setTransitionHooks(vnode, vnode.transition);  
 }  
 vnode.shapeFlag |= 512;  
 keys.delete(key);  
 keys.add(key);  
 } else {  
 keys.add(key);  
 if (max && keys.size > parseInt(max, 10)) {  
 pruneCacheEntry(keys.values().next().value);  
 }  
 }  
 vnode.shapeFlag |= 256;  
 current = vnode;  
 return isSuspense(rawVNode.type) ? rawVNode : vnode;  
 };  
 }  
};  
var KeepAlive = KeepAliveImpl;  
function matches(pattern, name) {  
 if (isArray(pattern)) {  
 return pattern.some((p2) => matches(p2, name));  
 } else if (isString(pattern)) {  
 return pattern.split(",").includes(name);  
 } else if (isRegExp(pattern)) {  
 return pattern.test(name);  
 }  
 return false;  
}  
function onActivated(hook, target) {  
 registerKeepAliveHook(hook, "a", target);  
}  
function onDeactivated(hook, target) {  
 registerKeepAliveHook(hook, "da", target);  
}  
function registerKeepAliveHook(hook, type, target = currentInstance) {  
 const wrappedHook = hook.\_\_wdc || (hook.\_\_wdc = () => {  
 let current = target;  
 while (current) {  
 if (current.isDeactivated) {  
 return;  
 }  
 current = current.parent;  
 }  
 return hook();  
 });  
 injectHook(type, wrappedHook, target);  
 if (target) {  
 let current = target.parent;  
 while (current && current.parent) {  
 if (isKeepAlive(current.parent.vnode)) {  
 injectToKeepAliveRoot(wrappedHook, type, target, current);  
 }  
 current = current.parent;  
 }  
 }  
}  
function injectToKeepAliveRoot(hook, type, target, keepAliveRoot) {  
 const injected = injectHook(  
 type,  
 hook,  
 keepAliveRoot,  
 true  
 );  
 onUnmounted(() => {  
 remove(keepAliveRoot[type], injected);  
 }, target);  
}  
function resetShapeFlag(vnode) {  
 vnode.shapeFlag &= ~256;  
 vnode.shapeFlag &= ~512;  
}  
function getInnerChild(vnode) {  
 return vnode.shapeFlag & 128 ? vnode.ssContent : vnode;  
}  
function injectHook(type, hook, target = currentInstance, prepend = false) {  
 if (target) {  
 const hooks = target[type] || (target[type] = []);  
 const wrappedHook = hook.\_\_weh || (hook.\_\_weh = (...args) => {  
 pauseTracking();  
 const reset = setCurrentInstance(target);  
 const res = callWithAsyncErrorHandling(hook, target, type, args);  
 reset();  
 resetTracking();  
 return res;  
 });  
 if (prepend) {  
 hooks.unshift(wrappedHook);  
 } else {  
 hooks.push(wrappedHook);  
 }  
 return wrappedHook;  
 } else if (true) {  
 const apiName = toHandlerKey(ErrorTypeStrings$1[type].replace(/ hook$/, ""));  
 warn$1(  
 `${apiName} is called when there is no active component instance to be associated with. Lifecycle injection APIs can only be used during execution of setup(). If you are using async setup(), make sure to register lifecycle hooks before the first await statement.`  
 );  
 }  
}  
var createHook = (lifecycle) => (hook, target = currentInstance) => {  
 if (!isInSSRComponentSetup || lifecycle === "sp") {  
 injectHook(lifecycle, (...args) => hook(...args), target);  
 }  
};  
var onBeforeMount = createHook("bm");  
var onMounted = createHook("m");  
var onBeforeUpdate = createHook("bu");  
var onUpdated = createHook("u");  
var onBeforeUnmount = createHook("bum");  
var onUnmounted = createHook("um");  
var onServerPrefetch = createHook("sp");  
var onRenderTriggered = createHook(  
 "rtg"  
);  
var onRenderTracked = createHook(  
 "rtc"  
);  
function onErrorCaptured(hook, target = currentInstance) {  
 injectHook("ec", hook, target);  
}  
var COMPONENTS = "components";  
var DIRECTIVES = "directives";  
function resolveComponent(name, maybeSelfReference) {  
 return resolveAsset(COMPONENTS, name, true, maybeSelfReference) || name;  
}  
var NULL\_DYNAMIC\_COMPONENT = Symbol.for("v-ndc");  
function resolveDynamicComponent(component) {  
 if (isString(component)) {  
 return resolveAsset(COMPONENTS, component, false) || component;  
 } else {  
 return component || NULL\_DYNAMIC\_COMPONENT;  
 }  
}  
function resolveDirective(name) {  
 return resolveAsset(DIRECTIVES, name);  
}  
function resolveAsset(type, name, warnMissing = true, maybeSelfReference = false) {  
 const instance = currentRenderingInstance || currentInstance;  
 if (instance) {  
 const Component = instance.type;  
 if (type === COMPONENTS) {  
 const selfName = getComponentName(  
 Component,  
 false  
 );  
 if (selfName && (selfName === name || selfName === camelize(name) || selfName === capitalize(camelize(name)))) {  
 return Component;  
 }  
 }  
 const res = (  
 resolve(instance[type] || Component[type], name) || // global registration  
 resolve(instance.appContext[type], name)  
 );  
 if (!res && maybeSelfReference) {  
 return Component;  
 }  
 if (warnMissing && !res) {  
 const extra = type === COMPONENTS ? `  
If this is a native custom element, make sure to exclude it from component resolution via compilerOptions.isCustomElement.` : ``;  
 warn$1(`Failed to resolve ${type.slice(0, -1)}: ${name}${extra}`);  
 }  
 return res;  
 } else if (true) {  
 warn$1(  
 `resolve${capitalize(type.slice(0, -1))} can only be used in render() or setup().`  
 );  
 }  
}  
function resolve(registry, name) {  
 return registry && (registry[name] || registry[camelize(name)] || registry[capitalize(camelize(name))]);  
}  
function renderList(source, renderItem, cache, index) {  
 let ret;  
 const cached = cache && cache[index];  
 if (isArray(source) || isString(source)) {  
 ret = new Array(source.length);  
 for (let i = 0, l = source.length; i < l; i++) {  
 ret[i] = renderItem(source[i], i, void 0, cached && cached[i]);  
 }  
 } else if (typeof source === "number") {  
 if (!Number.isInteger(source)) {  
 warn$1(`The v-for range expect an integer value but got ${source}.`);  
 }  
 ret = new Array(source);  
 for (let i = 0; i < source; i++) {  
 ret[i] = renderItem(i + 1, i, void 0, cached && cached[i]);  
 }  
 } else if (isObject(source)) {  
 if (source[Symbol.iterator]) {  
 ret = Array.from(  
 source,  
 (item, i) => renderItem(item, i, void 0, cached && cached[i])  
 );  
 } else {  
 const keys = Object.keys(source);  
 ret = new Array(keys.length);  
 for (let i = 0, l = keys.length; i < l; i++) {  
 const key = keys[i];  
 ret[i] = renderItem(source[key], key, i, cached && cached[i]);  
 }  
 }  
 } else {  
 ret = [];  
 }  
 if (cache) {  
 cache[index] = ret;  
 }  
 return ret;  
}  
function createSlots(slots, dynamicSlots) {  
 for (let i = 0; i < dynamicSlots.length; i++) {  
 const slot = dynamicSlots[i];  
 if (isArray(slot)) {  
 for (let j = 0; j < slot.length; j++) {  
 slots[slot[j].name] = slot[j].fn;  
 }  
 } else if (slot) {  
 slots[slot.name] = slot.key ? (...args) => {  
 const res = slot.fn(...args);  
 if (res) res.key = slot.key;  
 return res;  
 } : slot.fn;  
 }  
 }  
 return slots;  
}  
function renderSlot(slots, name, props = {}, fallback, noSlotted) {  
 if (currentRenderingInstance.isCE || currentRenderingInstance.parent && isAsyncWrapper(currentRenderingInstance.parent) && currentRenderingInstance.parent.isCE) {  
 if (name !== "default") props.name = name;  
 return createVNode("slot", props, fallback && fallback());  
 }  
 let slot = slots[name];  
 if (slot && slot.length > 1) {  
 warn$1(  
 `SSR-optimized slot function detected in a non-SSR-optimized render function. You need to mark this component with $dynamic-slots in the parent template.`  
 );  
 slot = () => [];  
 }  
 if (slot && slot.\_c) {  
 slot.\_d = false;  
 }  
 openBlock();  
 const validSlotContent = slot && ensureValidVNode(slot(props));  
 const rendered = createBlock(  
 Fragment,  
 {  
 key: (props.key || // slot content array of a dynamic conditional slot may have a branch  
 validSlotContent && validSlotContent.key || `\_${name}`) + // #7256 force differentiate fallback content from actual content  
 (!validSlotContent && fallback ? "\_fb" : "")  
 },  
 validSlotContent || (fallback ? fallback() : []),  
 validSlotContent && slots.\_ === 1 ? 64 : -2  
 );  
 if (!noSlotted && rendered.scopeId) {  
 rendered.slotScopeIds = [rendered.scopeId + "-s"];  
 }  
 if (slot && slot.\_c) {  
 slot.\_d = true;  
 }  
 return rendered;  
}  
function ensureValidVNode(vnodes) {  
 return vnodes.some((child) => {  
 if (!isVNode(child)) return true;  
 if (child.type === Comment) return false;  
 if (child.type === Fragment && !ensureValidVNode(child.children))  
 return false;  
 return true;  
 }) ? vnodes : null;  
}  
function toHandlers(obj, preserveCaseIfNecessary) {  
 const ret = {};  
 if (!isObject(obj)) {  
 warn$1(`v-on with no argument expects an object value.`);  
 return ret;  
 }  
 for (const key in obj) {  
 ret[preserveCaseIfNecessary && /[A-Z]/.test(key) ? `on:${key}` : toHandlerKey(key)] = obj[key];  
 }  
 return ret;  
}  
var getPublicInstance = (i) => {  
 if (!i) return null;  
 if (isStatefulComponent(i)) return getComponentPublicInstance(i);  
 return getPublicInstance(i.parent);  
};  
var publicPropertiesMap = (  
 extend(/\* @\_\_PURE\_\_ \*/ Object.create(null), {  
 $: (i) => i,  
 $el: (i) => i.vnode.el,  
 $data: (i) => i.data,  
 $props: (i) => true ? shallowReadonly(i.props) : i.props,  
 $attrs: (i) => true ? shallowReadonly(i.attrs) : i.attrs,  
 $slots: (i) => true ? shallowReadonly(i.slots) : i.slots,  
 $refs: (i) => true ? shallowReadonly(i.refs) : i.refs,  
 $parent: (i) => getPublicInstance(i.parent),  
 $root: (i) => getPublicInstance(i.root),  
 $emit: (i) => i.emit,  
 $options: (i) => \_\_VUE\_OPTIONS\_API\_\_ ? resolveMergedOptions(i) : i.type,  
 $forceUpdate: (i) => i.f || (i.f = () => {  
 i.effect.dirty = true;  
 queueJob(i.update);  
 }),  
 $nextTick: (i) => i.n || (i.n = nextTick.bind(i.proxy)),  
 $watch: (i) => \_\_VUE\_OPTIONS\_API\_\_ ? instanceWatch.bind(i) : NOOP  
 })  
);  
var isReservedPrefix = (key) => key === "\_" || key === "$";  
var hasSetupBinding = (state, key) => state !== EMPTY\_OBJ && !state.\_\_isScriptSetup && hasOwn(state, key);  
var PublicInstanceProxyHandlers = {  
 get({ \_: instance }, key) {  
 if (key === "\_\_v\_skip") {  
 return true;  
 }  
 const { ctx, setupState, data, props, accessCache, type, appContext } = instance;  
 if (key === "\_\_isVue") {  
 return true;  
 }  
 let normalizedProps;  
 if (key[0] !== "$") {  
 const n = accessCache[key];  
 if (n !== void 0) {  
 switch (n) {  
 case 1:  
 return setupState[key];  
 case 2:  
 return data[key];  
 case 4:  
 return ctx[key];  
 case 3:  
 return props[key];  
 }  
 } else if (hasSetupBinding(setupState, key)) {  
 accessCache[key] = 1;  
 return setupState[key];  
 } else if (data !== EMPTY\_OBJ && hasOwn(data, key)) {  
 accessCache[key] = 2;  
 return data[key];  
 } else if (  
 (normalizedProps = instance.propsOptions[0]) && hasOwn(normalizedProps, key)  
 ) {  
 accessCache[key] = 3;  
 return props[key];  
 } else if (ctx !== EMPTY\_OBJ && hasOwn(ctx, key)) {  
 accessCache[key] = 4;  
 return ctx[key];  
 } else if (!\_\_VUE\_OPTIONS\_API\_\_ || shouldCacheAccess) {  
 accessCache[key] = 0;  
 }  
 }  
 const publicGetter = publicPropertiesMap[key];  
 let cssModule, globalProperties;  
 if (publicGetter) {  
 if (key === "$attrs") {  
 track(instance.attrs, "get", "");  
 markAttrsAccessed();  
 } else if (key === "$slots") {  
 track(instance, "get", key);  
 }  
 return publicGetter(instance);  
 } else if (  
 (cssModule = type.\_\_cssModules) && (cssModule = cssModule[key])  
 ) {  
 return cssModule;  
 } else if (ctx !== EMPTY\_OBJ && hasOwn(ctx, key)) {  
 accessCache[key] = 4;  
 return ctx[key];  
 } else if (  
 globalProperties = appContext.config.globalProperties, hasOwn(globalProperties, key)  
 ) {  
 {  
 return globalProperties[key];  
 }  
 } else if (currentRenderingInstance && (!isString(key) || // #1091 avoid internal isRef/isVNode checks on component instance leading  
 key.indexOf("\_\_v") !== 0)) {  
 if (data !== EMPTY\_OBJ && isReservedPrefix(key[0]) && hasOwn(data, key)) {  
 warn$1(  
 `Property ${JSON.stringify(  
 key  
 )} must be accessed via $data because it starts with a reserved character ("$" or "\_") and is not proxied on the render context.`  
 );  
 } else if (instance === currentRenderingInstance) {  
 warn$1(  
 `Property ${JSON.stringify(key)} was accessed during render but is not defined on instance.`  
 );  
 }  
 }  
 },  
 set({ \_: instance }, key, value) {  
 const { data, setupState, ctx } = instance;  
 if (hasSetupBinding(setupState, key)) {  
 setupState[key] = value;  
 return true;  
 } else if (setupState.\_\_isScriptSetup && hasOwn(setupState, key)) {  
 warn$1(`Cannot mutate <script setup> binding "${key}" from Options API.`);  
 return false;  
 } else if (data !== EMPTY\_OBJ && hasOwn(data, key)) {  
 data[key] = value;  
 return true;  
 } else if (hasOwn(instance.props, key)) {  
 warn$1(`Attempting to mutate prop "${key}". Props are readonly.`);  
 return false;  
 }  
 if (key[0] === "$" && key.slice(1) in instance) {  
 warn$1(  
 `Attempting to mutate public property "${key}". Properties starting with $ are reserved and readonly.`  
 );  
 return false;  
 } else {  
 if (key in instance.appContext.config.globalProperties) {  
 Object.defineProperty(ctx, key, {  
 enumerable: true,  
 configurable: true,  
 value  
 });  
 } else {  
 ctx[key] = value;  
 }  
 }  
 return true;  
 },  
 has({  
 \_: { data, setupState, accessCache, ctx, appContext, propsOptions }  
 }, key) {  
 let normalizedProps;  
 return !!accessCache[key] || data !== EMPTY\_OBJ && hasOwn(data, key) || hasSetupBinding(setupState, key) || (normalizedProps = propsOptions[0]) && hasOwn(normalizedProps, key) || hasOwn(ctx, key) || hasOwn(publicPropertiesMap, key) || hasOwn(appContext.config.globalProperties, key);  
 },  
 defineProperty(target, key, descriptor) {  
 if (descriptor.get != null) {  
 target.\_.accessCache[key] = 0;  
 } else if (hasOwn(descriptor, "value")) {  
 this.set(target, key, descriptor.value, null);  
 }  
 return Reflect.defineProperty(target, key, descriptor);  
 }  
};  
if (true) {  
 PublicInstanceProxyHandlers.ownKeys = (target) => {  
 warn$1(  
 `Avoid app logic that relies on enumerating keys on a component instance. The keys will be empty in production mode to avoid performance overhead.`  
 );  
 return Reflect.ownKeys(target);  
 };  
}  
var RuntimeCompiledPublicInstanceProxyHandlers = extend(  
 {},  
 PublicInstanceProxyHandlers,  
 {  
 get(target, key) {  
 if (key === Symbol.unscopables) {  
 return;  
 }  
 return PublicInstanceProxyHandlers.get(target, key, target);  
 },  
 has(\_, key) {  
 const has2 = key[0] !== "\_" && !isGloballyAllowed(key);  
 if (!has2 && PublicInstanceProxyHandlers.has(\_, key)) {  
 warn$1(  
 `Property ${JSON.stringify(  
 key  
 )} should not start with \_ which is a reserved prefix for Vue internals.`  
 );  
 }  
 return has2;  
 }  
 }  
);  
function createDevRenderContext(instance) {  
 const target = {};  
 Object.defineProperty(target, `\_`, {  
 configurable: true,  
 enumerable: false,  
 get: () => instance  
 });  
 Object.keys(publicPropertiesMap).forEach((key) => {  
 Object.defineProperty(target, key, {  
 configurable: true,  
 enumerable: false,  
 get: () => publicPropertiesMap[key](instance),  
 set: NOOP  
 });  
 });  
 return target;  
}  
function exposePropsOnRenderContext(instance) {  
 const {  
 ctx,  
 propsOptions: [propsOptions]  
 } = instance;  
 if (propsOptions) {  
 Object.keys(propsOptions).forEach((key) => {  
 Object.defineProperty(ctx, key, {  
 enumerable: true,  
 configurable: true,  
 get: () => instance.props[key],  
 set: NOOP  
 });  
 });  
 }  
}  
function exposeSetupStateOnRenderContext(instance) {  
 const { ctx, setupState } = instance;  
 Object.keys(toRaw(setupState)).forEach((key) => {  
 if (!setupState.\_\_isScriptSetup) {  
 if (isReservedPrefix(key[0])) {  
 warn$1(  
 `setup() return property ${JSON.stringify(  
 key  
 )} should not start with "$" or "\_" which are reserved prefixes for Vue internals.`  
 );  
 return;  
 }  
 Object.defineProperty(ctx, key, {  
 enumerable: true,  
 configurable: true,  
 get: () => setupState[key],  
 set: NOOP  
 });  
 }  
 });  
}  
var warnRuntimeUsage = (method) => warn$1(  
 `${method}() is a compiler-hint helper that is only usable inside <script setup> of a single file component. Its arguments should be compiled away and passing it at runtime has no effect.`  
);  
function defineProps() {  
 if (true) {  
 warnRuntimeUsage(`defineProps`);  
 }  
 return null;  
}  
function defineEmits() {  
 if (true) {  
 warnRuntimeUsage(`defineEmits`);  
 }  
 return null;  
}  
function defineExpose(exposed) {  
 if (true) {  
 warnRuntimeUsage(`defineExpose`);  
 }  
}  
function defineOptions(options) {  
 if (true) {  
 warnRuntimeUsage(`defineOptions`);  
 }  
}  
function defineSlots() {  
 if (true) {  
 warnRuntimeUsage(`defineSlots`);  
 }  
 return null;  
}  
function defineModel() {  
 if (true) {  
 warnRuntimeUsage("defineModel");  
 }  
}  
function withDefaults(props, defaults) {  
 if (true) {  
 warnRuntimeUsage(`withDefaults`);  
 }  
 return null;  
}  
function useSlots() {  
 return getContext().slots;  
}  
function useAttrs() {  
 return getContext().attrs;  
}  
function getContext() {  
 const i = getCurrentInstance();  
 if (!i) {  
 warn$1(`useContext() called without active instance.`);  
 }  
 return i.setupContext || (i.setupContext = createSetupContext(i));  
}  
function normalizePropsOrEmits(props) {  
 return isArray(props) ? props.reduce(  
 (normalized, p2) => (normalized[p2] = null, normalized),  
 {}  
 ) : props;  
}  
function mergeDefaults(raw, defaults) {  
 const props = normalizePropsOrEmits(raw);  
 for (const key in defaults) {  
 if (key.startsWith("\_\_skip")) continue;  
 let opt = props[key];  
 if (opt) {  
 if (isArray(opt) || isFunction(opt)) {  
 opt = props[key] = { type: opt, default: defaults[key] };  
 } else {  
 opt.default = defaults[key];  
 }  
 } else if (opt === null) {  
 opt = props[key] = { default: defaults[key] };  
 } else if (true) {  
 warn$1(`props default key "${key}" has no corresponding declaration.`);  
 }  
 if (opt && defaults[`\_\_skip\_${key}`]) {  
 opt.skipFactory = true;  
 }  
 }  
 return props;  
}  
function mergeModels(a, b) {  
 if (!a || !b) return a || b;  
 if (isArray(a) && isArray(b)) return a.concat(b);  
 return extend({}, normalizePropsOrEmits(a), normalizePropsOrEmits(b));  
}  
function createPropsRestProxy(props, excludedKeys) {  
 const ret = {};  
 for (const key in props) {  
 if (!excludedKeys.includes(key)) {  
 Object.defineProperty(ret, key, {  
 enumerable: true,  
 get: () => props[key]  
 });  
 }  
 }  
 return ret;  
}  
function withAsyncContext(getAwaitable) {  
 const ctx = getCurrentInstance();  
 if (!ctx) {  
 warn$1(  
 `withAsyncContext called without active current instance. This is likely a bug.`  
 );  
 }  
 let awaitable = getAwaitable();  
 unsetCurrentInstance();  
 if (isPromise(awaitable)) {  
 awaitable = awaitable.catch((e) => {  
 setCurrentInstance(ctx);  
 throw e;  
 });  
 }  
 return [awaitable, () => setCurrentInstance(ctx)];  
}  
function createDuplicateChecker() {  
 const cache = /\* @\_\_PURE\_\_ \*/ Object.create(null);  
 return (type, key) => {  
 if (cache[key]) {  
 warn$1(`${type} property "${key}" is already defined in ${cache[key]}.`);  
 } else {  
 cache[key] = type;  
 }  
 };  
}  
var shouldCacheAccess = true;  
function applyOptions(instance) {  
 const options = resolveMergedOptions(instance);  
 const publicThis = instance.proxy;  
 const ctx = instance.ctx;  
 shouldCacheAccess = false;  
 if (options.beforeCreate) {  
 callHook(options.beforeCreate, instance, "bc");  
 }  
 const {  
 data: dataOptions,  
 computed: computedOptions,  
 methods,  
 watch: watchOptions,  
 provide: provideOptions,  
 inject: injectOptions,  
 created,  
 beforeMount,  
 mounted,  
 beforeUpdate,  
 updated,  
 activated,  
 deactivated,  
 beforeDestroy,  
 beforeUnmount,  
 destroyed,  
 unmounted,  
 render: render2,  
 renderTracked,  
 renderTriggered,  
 errorCaptured,  
 serverPrefetch,  
 expose,  
 inheritAttrs,  
 components,  
 directives,  
 filters  
 } = options;  
 const checkDuplicateProperties = true ? createDuplicateChecker() : null;  
 if (true) {  
 const [propsOptions] = instance.propsOptions;  
 if (propsOptions) {  
 for (const key in propsOptions) {  
 checkDuplicateProperties("Props", key);  
 }  
 }  
 }  
 if (injectOptions) {  
 resolveInjections(injectOptions, ctx, checkDuplicateProperties);  
 }  
 if (methods) {  
 for (const key in methods) {  
 const methodHandler = methods[key];  
 if (isFunction(methodHandler)) {  
 if (true) {  
 Object.defineProperty(ctx, key, {  
 value: methodHandler.bind(publicThis),  
 configurable: true,  
 enumerable: true,  
 writable: true  
 });  
 } else {  
 ctx[key] = methodHandler.bind(publicThis);  
 }  
 if (true) {  
 checkDuplicateProperties("Methods", key);  
 }  
 } else if (true) {  
 warn$1(  
 `Method "${key}" has type "${typeof methodHandler}" in the component definition. Did you reference the function correctly?`  
 );  
 }  
 }  
 }  
 if (dataOptions) {  
 if (!isFunction(dataOptions)) {  
 warn$1(  
 `The data option must be a function. Plain object usage is no longer supported.`  
 );  
 }  
 const data = dataOptions.call(publicThis, publicThis);  
 if (isPromise(data)) {  
 warn$1(  
 `data() returned a Promise - note data() cannot be async; If you intend to perform data fetching before component renders, use async setup() + <Suspense>.`  
 );  
 }  
 if (!isObject(data)) {  
 warn$1(`data() should return an object.`);  
 } else {  
 instance.data = reactive(data);  
 if (true) {  
 for (const key in data) {  
 checkDuplicateProperties("Data", key);  
 if (!isReservedPrefix(key[0])) {  
 Object.defineProperty(ctx, key, {  
 configurable: true,  
 enumerable: true,  
 get: () => data[key],  
 set: NOOP  
 });  
 }  
 }  
 }  
 }  
 }  
 shouldCacheAccess = true;  
 if (computedOptions) {  
 for (const key in computedOptions) {  
 const opt = computedOptions[key];  
 const get2 = isFunction(opt) ? opt.bind(publicThis, publicThis) : isFunction(opt.get) ? opt.get.bind(publicThis, publicThis) : NOOP;  
 if (get2 === NOOP) {  
 warn$1(`Computed property "${key}" has no getter.`);  
 }  
 const set2 = !isFunction(opt) && isFunction(opt.set) ? opt.set.bind(publicThis) : true ? () => {  
 warn$1(  
 `Write operation failed: computed property "${key}" is readonly.`  
 );  
 } : NOOP;  
 const c = computed2({  
 get: get2,  
 set: set2  
 });  
 Object.defineProperty(ctx, key, {  
 enumerable: true,  
 configurable: true,  
 get: () => c.value,  
 set: (v) => c.value = v  
 });  
 if (true) {  
 checkDuplicateProperties("Computed", key);  
 }  
 }  
 }  
 if (watchOptions) {  
 for (const key in watchOptions) {  
 createWatcher(watchOptions[key], ctx, publicThis, key);  
 }  
 }  
 if (provideOptions) {  
 const provides = isFunction(provideOptions) ? provideOptions.call(publicThis) : provideOptions;  
 Reflect.ownKeys(provides).forEach((key) => {  
 provide(key, provides[key]);  
 });  
 }  
 if (created) {  
 callHook(created, instance, "c");  
 }  
 function registerLifecycleHook(register, hook) {  
 if (isArray(hook)) {  
 hook.forEach((\_hook) => register(\_hook.bind(publicThis)));  
 } else if (hook) {  
 register(hook.bind(publicThis));  
 }  
 }  
 registerLifecycleHook(onBeforeMount, beforeMount);  
 registerLifecycleHook(onMounted, mounted);  
 registerLifecycleHook(onBeforeUpdate, beforeUpdate);  
 registerLifecycleHook(onUpdated, updated);  
 registerLifecycleHook(onActivated, activated);  
 registerLifecycleHook(onDeactivated, deactivated);  
 registerLifecycleHook(onErrorCaptured, errorCaptured);  
 registerLifecycleHook(onRenderTracked, renderTracked);  
 registerLifecycleHook(onRenderTriggered, renderTriggered);  
 registerLifecycleHook(onBeforeUnmount, beforeUnmount);  
 registerLifecycleHook(onUnmounted, unmounted);  
 registerLifecycleHook(onServerPrefetch, serverPrefetch);  
 if (isArray(expose)) {  
 if (expose.length) {  
 const exposed = instance.exposed || (instance.exposed = {});  
 expose.forEach((key) => {  
 Object.defineProperty(exposed, key, {  
 get: () => publicThis[key],  
 set: (val) => publicThis[key] = val  
 });  
 });  
 } else if (!instance.exposed) {  
 instance.exposed = {};  
 }  
 }  
 if (render2 && instance.render === NOOP) {  
 instance.render = render2;  
 }  
 if (inheritAttrs != null) {  
 instance.inheritAttrs = inheritAttrs;  
 }  
 if (components) instance.components = components;  
 if (directives) instance.directives = directives;  
}  
function resolveInjections(injectOptions, ctx, checkDuplicateProperties = NOOP) {  
 if (isArray(injectOptions)) {  
 injectOptions = normalizeInject(injectOptions);  
 }  
 for (const key in injectOptions) {  
 const opt = injectOptions[key];  
 let injected;  
 if (isObject(opt)) {  
 if ("default" in opt) {  
 injected = inject(  
 opt.from || key,  
 opt.default,  
 true  
 );  
 } else {  
 injected = inject(opt.from || key);  
 }  
 } else {  
 injected = inject(opt);  
 }  
 if (isRef2(injected)) {  
 Object.defineProperty(ctx, key, {  
 enumerable: true,  
 configurable: true,  
 get: () => injected.value,  
 set: (v) => injected.value = v  
 });  
 } else {  
 ctx[key] = injected;  
 }  
 if (true) {  
 checkDuplicateProperties("Inject", key);  
 }  
 }  
}  
function callHook(hook, instance, type) {  
 callWithAsyncErrorHandling(  
 isArray(hook) ? hook.map((h2) => h2.bind(instance.proxy)) : hook.bind(instance.proxy),  
 instance,  
 type  
 );  
}  
function createWatcher(raw, ctx, publicThis, key) {  
 const getter = key.includes(".") ? createPathGetter(publicThis, key) : () => publicThis[key];  
 if (isString(raw)) {  
 const handler = ctx[raw];  
 if (isFunction(handler)) {  
 watch(getter, handler);  
 } else if (true) {  
 warn$1(`Invalid watch handler specified by key "${raw}"`, handler);  
 }  
 } else if (isFunction(raw)) {  
 watch(getter, raw.bind(publicThis));  
 } else if (isObject(raw)) {  
 if (isArray(raw)) {  
 raw.forEach((r) => createWatcher(r, ctx, publicThis, key));  
 } else {  
 const handler = isFunction(raw.handler) ? raw.handler.bind(publicThis) : ctx[raw.handler];  
 if (isFunction(handler)) {  
 watch(getter, handler, raw);  
 } else if (true) {  
 warn$1(`Invalid watch handler specified by key "${raw.handler}"`, handler);  
 }  
 }  
 } else if (true) {  
 warn$1(`Invalid watch option: "${key}"`, raw);  
 }  
}  
function resolveMergedOptions(instance) {  
 const base = instance.type;  
 const { mixins, extends: extendsOptions } = base;  
 const {  
 mixins: globalMixins,  
 optionsCache: cache,  
 config: { optionMergeStrategies }  
 } = instance.appContext;  
 const cached = cache.get(base);  
 let resolved;  
 if (cached) {  
 resolved = cached;  
 } else if (!globalMixins.length && !mixins && !extendsOptions) {  
 {  
 resolved = base;  
 }  
 } else {  
 resolved = {};  
 if (globalMixins.length) {  
 globalMixins.forEach(  
 (m) => mergeOptions(resolved, m, optionMergeStrategies, true)  
 );  
 }  
 mergeOptions(resolved, base, optionMergeStrategies);  
 }  
 if (isObject(base)) {  
 cache.set(base, resolved);  
 }  
 return resolved;  
}  
function mergeOptions(to, from, strats, asMixin = false) {  
 const { mixins, extends: extendsOptions } = from;  
 if (extendsOptions) {  
 mergeOptions(to, extendsOptions, strats, true);  
 }  
 if (mixins) {  
 mixins.forEach(  
 (m) => mergeOptions(to, m, strats, true)  
 );  
 }  
 for (const key in from) {  
 if (asMixin && key === "expose") {  
 warn$1(  
 `"expose" option is ignored when declared in mixins or extends. It should only be declared in the base component itself.`  
 );  
 } else {  
 const strat = internalOptionMergeStrats[key] || strats && strats[key];  
 to[key] = strat ? strat(to[key], from[key]) : from[key];  
 }  
 }  
 return to;  
}  
var internalOptionMergeStrats = {  
 data: mergeDataFn,  
 props: mergeEmitsOrPropsOptions,  
 emits: mergeEmitsOrPropsOptions,  
 methods: mergeObjectOptions,  
 computed: mergeObjectOptions,  
 beforeCreate: mergeAsArray,  
 created: mergeAsArray,  
 beforeMount: mergeAsArray,  
 mounted: mergeAsArray,  
 beforeUpdate: mergeAsArray,  
 updated: mergeAsArray,  
 beforeDestroy: mergeAsArray,  
 beforeUnmount: mergeAsArray,  
 destroyed: mergeAsArray,  
 unmounted: mergeAsArray,  
 activated: mergeAsArray,  
 deactivated: mergeAsArray,  
 errorCaptured: mergeAsArray,  
 serverPrefetch: mergeAsArray,  
 components: mergeObjectOptions,  
 directives: mergeObjectOptions,  
 watch: mergeWatchOptions,  
 provide: mergeDataFn,  
 inject: mergeInject  
};  
function mergeDataFn(to, from) {  
 if (!from) {  
 return to;  
 }  
 if (!to) {  
 return from;  
 }  
 return function mergedDataFn() {  
 return extend(  
 isFunction(to) ? to.call(this, this) : to,  
 isFunction(from) ? from.call(this, this) : from  
 );  
 };  
}  
function mergeInject(to, from) {  
 return mergeObjectOptions(normalizeInject(to), normalizeInject(from));  
}  
function normalizeInject(raw) {  
 if (isArray(raw)) {  
 const res = {};  
 for (let i = 0; i < raw.length; i++) {  
 res[raw[i]] = raw[i];  
 }  
 return res;  
 }  
 return raw;  
}  
function mergeAsArray(to, from) {  
 return to ? [...new Set([].concat(to, from))] : from;  
}  
function mergeObjectOptions(to, from) {  
 return to ? extend(/\* @\_\_PURE\_\_ \*/ Object.create(null), to, from) : from;  
}  
function mergeEmitsOrPropsOptions(to, from) {  
 if (to) {  
 if (isArray(to) && isArray(from)) {  
 return [.../\* @\_\_PURE\_\_ \*/ new Set([...to, ...from])];  
 }  
 return extend(  
 normalizePropsOrEmits(to),  
 normalizePropsOrEmits(from != null ? from : {})  
 );  
 } else {  
 return from;  
 }  
}  
function mergeWatchOptions(to, from) {  
 if (!to) return from;  
 if (!from) return to;  
 const merged = extend(/\* @\_\_PURE\_\_ \*/ Object.create(null), to);  
 for (const key in from) {  
 merged[key] = mergeAsArray(to[key], from[key]);  
 }  
 return merged;  
}  
function createAppContext() {  
 return {  
 app: null,  
 config: {  
 isNativeTag: NO,  
 performance: false,  
 globalProperties: {},  
 optionMergeStrategies: {},  
 errorHandler: void 0,  
 warnHandler: void 0,  
 compilerOptions: {}  
 },  
 mixins: [],  
 components: {},  
 directives: {},  
 provides: /\* @\_\_PURE\_\_ \*/ Object.create(null),  
 optionsCache: /\* @\_\_PURE\_\_ \*/ new WeakMap(),  
 propsCache: /\* @\_\_PURE\_\_ \*/ new WeakMap(),  
 emitsCache: /\* @\_\_PURE\_\_ \*/ new WeakMap()  
 };  
}  
var uid$1 = 0;  
function createAppAPI(render2, hydrate2) {  
 return function createApp2(rootComponent, rootProps = null) {  
 if (!isFunction(rootComponent)) {  
 rootComponent = extend({}, rootComponent);  
 }  
 if (rootProps != null && !isObject(rootProps)) {  
 warn$1(`root props passed to app.mount() must be an object.`);  
 rootProps = null;  
 }  
 const context = createAppContext();  
 const installedPlugins = /\* @\_\_PURE\_\_ \*/ new WeakSet();  
 let isMounted = false;  
 const app = context.app = {  
 \_uid: uid$1++,  
 \_component: rootComponent,  
 \_props: rootProps,  
 \_container: null,  
 \_context: context,  
 \_instance: null,  
 version,  
 get config() {  
 return context.config;  
 },  
 set config(v) {  
 if (true) {  
 warn$1(  
 `app.config cannot be replaced. Modify individual options instead.`  
 );  
 }  
 },  
 use(plugin, ...options) {  
 if (installedPlugins.has(plugin)) {  
 warn$1(`Plugin has already been applied to target app.`);  
 } else if (plugin && isFunction(plugin.install)) {  
 installedPlugins.add(plugin);  
 plugin.install(app, ...options);  
 } else if (isFunction(plugin)) {  
 installedPlugins.add(plugin);  
 plugin(app, ...options);  
 } else if (true) {  
 warn$1(  
 `A plugin must either be a function or an object with an "install" function.`  
 );  
 }  
 return app;  
 },  
 mixin(mixin) {  
 if (\_\_VUE\_OPTIONS\_API\_\_) {  
 if (!context.mixins.includes(mixin)) {  
 context.mixins.push(mixin);  
 } else if (true) {  
 warn$1(  
 "Mixin has already been applied to target app" + (mixin.name ? `: ${mixin.name}` : "")  
 );  
 }  
 } else if (true) {  
 warn$1("Mixins are only available in builds supporting Options API");  
 }  
 return app;  
 },  
 component(name, component) {  
 if (true) {  
 validateComponentName(name, context.config);  
 }  
 if (!component) {  
 return context.components[name];  
 }  
 if (context.components[name]) {  
 warn$1(`Component "${name}" has already been registered in target app.`);  
 }  
 context.components[name] = component;  
 return app;  
 },  
 directive(name, directive) {  
 if (true) {  
 validateDirectiveName(name);  
 }  
 if (!directive) {  
 return context.directives[name];  
 }  
 if (context.directives[name]) {  
 warn$1(`Directive "${name}" has already been registered in target app.`);  
 }  
 context.directives[name] = directive;  
 return app;  
 },  
 mount(rootContainer, isHydrate, namespace) {  
 if (!isMounted) {  
 if (rootContainer.\_\_vue\_app\_\_) {  
 warn$1(  
 `There is already an app instance mounted on the host container.  
 If you want to mount another app on the same host container, you need to unmount the previous app by calling \`app.unmount()\` first.`  
 );  
 }  
 const vnode = createVNode(rootComponent, rootProps);  
 vnode.appContext = context;  
 if (namespace === true) {  
 namespace = "svg";  
 } else if (namespace === false) {  
 namespace = void 0;  
 }  
 if (true) {  
 context.reload = () => {  
 render2(  
 cloneVNode(vnode),  
 rootContainer,  
 namespace  
 );  
 };  
 }  
 if (isHydrate && hydrate2) {  
 hydrate2(vnode, rootContainer);  
 } else {  
 render2(vnode, rootContainer, namespace);  
 }  
 isMounted = true;  
 app.\_container = rootContainer;  
 rootContainer.\_\_vue\_app\_\_ = app;  
 if (true) {  
 app.\_instance = vnode.component;  
 devtoolsInitApp(app, version);  
 }  
 return getComponentPublicInstance(vnode.component);  
 } else if (true) {  
 warn$1(  
 `App has already been mounted.  
If you want to remount the same app, move your app creation logic into a factory function and create fresh app instances for each mount - e.g. \`const createMyApp = () => createApp(App)\``  
 );  
 }  
 },  
 unmount() {  
 if (isMounted) {  
 render2(null, app.\_container);  
 if (true) {  
 app.\_instance = null;  
 devtoolsUnmountApp(app);  
 }  
 delete app.\_container.\_\_vue\_app\_\_;  
 } else if (true) {  
 warn$1(`Cannot unmount an app that is not mounted.`);  
 }  
 },  
 provide(key, value) {  
 if (key in context.provides) {  
 warn$1(  
 `App already provides property with key "${String(key)}". It will be overwritten with the new value.`  
 );  
 }  
 context.provides[key] = value;  
 return app;  
 },  
 runWithContext(fn) {  
 const lastApp = currentApp;  
 currentApp = app;  
 try {  
 return fn();  
 } finally {  
 currentApp = lastApp;  
 }  
 }  
 };  
 return app;  
 };  
}  
var currentApp = null;  
function provide(key, value) {  
 if (!currentInstance) {  
 if (true) {  
 warn$1(`provide() can only be used inside setup().`);  
 }  
 } else {  
 let provides = currentInstance.provides;  
 const parentProvides = currentInstance.parent && currentInstance.parent.provides;  
 if (parentProvides === provides) {  
 provides = currentInstance.provides = Object.create(parentProvides);  
 }  
 provides[key] = value;  
 }  
}  
function inject(key, defaultValue, treatDefaultAsFactory = false) {  
 const instance = currentInstance || currentRenderingInstance;  
 if (instance || currentApp) {  
 const provides = instance ? instance.parent == null ? instance.vnode.appContext && instance.vnode.appContext.provides : instance.parent.provides : currentApp.\_context.provides;  
 if (provides && key in provides) {  
 return provides[key];  
 } else if (arguments.length > 1) {  
 return treatDefaultAsFactory && isFunction(defaultValue) ? defaultValue.call(instance && instance.proxy) : defaultValue;  
 } else if (true) {  
 warn$1(`injection "${String(key)}" not found.`);  
 }  
 } else if (true) {  
 warn$1(`inject() can only be used inside setup() or functional components.`);  
 }  
}  
function hasInjectionContext() {  
 return !!(currentInstance || currentRenderingInstance || currentApp);  
}  
var internalObjectProto = {};  
var createInternalObject = () => Object.create(internalObjectProto);  
var isInternalObject = (obj) => Object.getPrototypeOf(obj) === internalObjectProto;  
function initProps(instance, rawProps, isStateful, isSSR = false) {  
 const props = {};  
 const attrs = createInternalObject();  
 instance.propsDefaults = /\* @\_\_PURE\_\_ \*/ Object.create(null);  
 setFullProps(instance, rawProps, props, attrs);  
 for (const key in instance.propsOptions[0]) {  
 if (!(key in props)) {  
 props[key] = void 0;  
 }  
 }  
 if (true) {  
 validateProps(rawProps || {}, props, instance);  
 }  
 if (isStateful) {  
 instance.props = isSSR ? props : shallowReactive(props);  
 } else {  
 if (!instance.type.props) {  
 instance.props = attrs;  
 } else {  
 instance.props = props;  
 }  
 }  
 instance.attrs = attrs;  
}  
function isInHmrContext(instance) {  
 while (instance) {  
 if (instance.type.\_\_hmrId) return true;  
 instance = instance.parent;  
 }  
}  
function updateProps(instance, rawProps, rawPrevProps, optimized) {  
 const {  
 props,  
 attrs,  
 vnode: { patchFlag }  
 } = instance;  
 const rawCurrentProps = toRaw(props);  
 const [options] = instance.propsOptions;  
 let hasAttrsChanged = false;  
 if (  
 !isInHmrContext(instance) && (optimized || patchFlag > 0) && !(patchFlag & 16)  
 ) {  
 if (patchFlag & 8) {  
 const propsToUpdate = instance.vnode.dynamicProps;  
 for (let i = 0; i < propsToUpdate.length; i++) {  
 let key = propsToUpdate[i];  
 if (isEmitListener(instance.emitsOptions, key)) {  
 continue;  
 }  
 const value = rawProps[key];  
 if (options) {  
 if (hasOwn(attrs, key)) {  
 if (value !== attrs[key]) {  
 attrs[key] = value;  
 hasAttrsChanged = true;  
 }  
 } else {  
 const camelizedKey = camelize(key);  
 props[camelizedKey] = resolvePropValue(  
 options,  
 rawCurrentProps,  
 camelizedKey,  
 value,  
 instance,  
 false  
 );  
 }  
 } else {  
 if (value !== attrs[key]) {  
 attrs[key] = value;  
 hasAttrsChanged = true;  
 }  
 }  
 }  
 }  
 } else {  
 if (setFullProps(instance, rawProps, props, attrs)) {  
 hasAttrsChanged = true;  
 }  
 let kebabKey;  
 for (const key in rawCurrentProps) {  
 if (!rawProps || // for camelCase  
 !hasOwn(rawProps, key) && // it's possible the original props was passed in as kebab-case  
 ((kebabKey = hyphenate(key)) === key || !hasOwn(rawProps, kebabKey))) {  
 if (options) {  
 if (rawPrevProps && // for camelCase  
 (rawPrevProps[key] !== void 0 || // for kebab-case  
 rawPrevProps[kebabKey] !== void 0)) {  
 props[key] = resolvePropValue(  
 options,  
 rawCurrentProps,  
 key,  
 void 0,  
 instance,  
 true  
 );  
 }  
 } else {  
 delete props[key];  
 }  
 }  
 }  
 if (attrs !== rawCurrentProps) {  
 for (const key in attrs) {  
 if (!rawProps || !hasOwn(rawProps, key) && true) {  
 delete attrs[key];  
 hasAttrsChanged = true;  
 }  
 }  
 }  
 }  
 if (hasAttrsChanged) {  
 trigger(instance.attrs, "set", "");  
 }  
 if (true) {  
 validateProps(rawProps || {}, props, instance);  
 }  
}  
function setFullProps(instance, rawProps, props, attrs) {  
 const [options, needCastKeys] = instance.propsOptions;  
 let hasAttrsChanged = false;  
 let rawCastValues;  
 if (rawProps) {  
 for (let key in rawProps) {  
 if (isReservedProp(key)) {  
 continue;  
 }  
 const value = rawProps[key];  
 let camelKey;  
 if (options && hasOwn(options, camelKey = camelize(key))) {  
 if (!needCastKeys || !needCastKeys.includes(camelKey)) {  
 props[camelKey] = value;  
 } else {  
 (rawCastValues || (rawCastValues = {}))[camelKey] = value;  
 }  
 } else if (!isEmitListener(instance.emitsOptions, key)) {  
 if (!(key in attrs) || value !== attrs[key]) {  
 attrs[key] = value;  
 hasAttrsChanged = true;  
 }  
 }  
 }  
 }  
 if (needCastKeys) {  
 const rawCurrentProps = toRaw(props);  
 const castValues = rawCastValues || EMPTY\_OBJ;  
 for (let i = 0; i < needCastKeys.length; i++) {  
 const key = needCastKeys[i];  
 props[key] = resolvePropValue(  
 options,  
 rawCurrentProps,  
 key,  
 castValues[key],  
 instance,  
 !hasOwn(castValues, key)  
 );  
 }  
 }  
 return hasAttrsChanged;  
}  
function resolvePropValue(options, props, key, value, instance, isAbsent) {  
 const opt = options[key];  
 if (opt != null) {  
 const hasDefault = hasOwn(opt, "default");  
 if (hasDefault && value === void 0) {  
 const defaultValue = opt.default;  
 if (opt.type !== Function && !opt.skipFactory && isFunction(defaultValue)) {  
 const { propsDefaults } = instance;  
 if (key in propsDefaults) {  
 value = propsDefaults[key];  
 } else {  
 const reset = setCurrentInstance(instance);  
 value = propsDefaults[key] = defaultValue.call(  
 null,  
 props  
 );  
 reset();  
 }  
 } else {  
 value = defaultValue;  
 }  
 }  
 if (opt[  
 0  
 ]) {  
 if (isAbsent && !hasDefault) {  
 value = false;  
 } else if (opt[  
 1  
 ] && (value === "" || value === hyphenate(key))) {  
 value = true;  
 }  
 }  
 }  
 return value;  
}  
var mixinPropsCache = /\* @\_\_PURE\_\_ \*/ new WeakMap();  
function normalizePropsOptions(comp, appContext, asMixin = false) {  
 const cache = \_\_VUE\_OPTIONS\_API\_\_ && asMixin ? mixinPropsCache : appContext.propsCache;  
 const cached = cache.get(comp);  
 if (cached) {  
 return cached;  
 }  
 const raw = comp.props;  
 const normalized = {};  
 const needCastKeys = [];  
 let hasExtends = false;  
 if (\_\_VUE\_OPTIONS\_API\_\_ && !isFunction(comp)) {  
 const extendProps = (raw2) => {  
 hasExtends = true;  
 const [props, keys] = normalizePropsOptions(raw2, appContext, true);  
 extend(normalized, props);  
 if (keys) needCastKeys.push(...keys);  
 };  
 if (!asMixin && appContext.mixins.length) {  
 appContext.mixins.forEach(extendProps);  
 }  
 if (comp.extends) {  
 extendProps(comp.extends);  
 }  
 if (comp.mixins) {  
 comp.mixins.forEach(extendProps);  
 }  
 }  
 if (!raw && !hasExtends) {  
 if (isObject(comp)) {  
 cache.set(comp, EMPTY\_ARR);  
 }  
 return EMPTY\_ARR;  
 }  
 if (isArray(raw)) {  
 for (let i = 0; i < raw.length; i++) {  
 if (!isString(raw[i])) {  
 warn$1(`props must be strings when using array syntax.`, raw[i]);  
 }  
 const normalizedKey = camelize(raw[i]);  
 if (validatePropName(normalizedKey)) {  
 normalized[normalizedKey] = EMPTY\_OBJ;  
 }  
 }  
 } else if (raw) {  
 if (!isObject(raw)) {  
 warn$1(`invalid props options`, raw);  
 }  
 for (const key in raw) {  
 const normalizedKey = camelize(key);  
 if (validatePropName(normalizedKey)) {  
 const opt = raw[key];  
 const prop = normalized[normalizedKey] = isArray(opt) || isFunction(opt) ? { type: opt } : extend({}, opt);  
 if (prop) {  
 const booleanIndex = getTypeIndex(Boolean, prop.type);  
 const stringIndex = getTypeIndex(String, prop.type);  
 prop[  
 0  
 ] = booleanIndex > -1;  
 prop[  
 1  
 ] = stringIndex < 0 || booleanIndex < stringIndex;  
 if (booleanIndex > -1 || hasOwn(prop, "default")) {  
 needCastKeys.push(normalizedKey);  
 }  
 }  
 }  
 }  
 }  
 const res = [normalized, needCastKeys];  
 if (isObject(comp)) {  
 cache.set(comp, res);  
 }  
 return res;  
}  
function validatePropName(key) {  
 if (key[0] !== "$" && !isReservedProp(key)) {  
 return true;  
 } else if (true) {  
 warn$1(`Invalid prop name: "${key}" is a reserved property.`);  
 }  
 return false;  
}  
function getType(ctor) {  
 if (ctor === null) {  
 return "null";  
 }  
 if (typeof ctor === "function") {  
 return ctor.name || "";  
 } else if (typeof ctor === "object") {  
 const name = ctor.constructor && ctor.constructor.name;  
 return name || "";  
 }  
 return "";  
}  
function isSameType(a, b) {  
 return getType(a) === getType(b);  
}  
function getTypeIndex(type, expectedTypes) {  
 if (isArray(expectedTypes)) {  
 return expectedTypes.findIndex((t) => isSameType(t, type));  
 } else if (isFunction(expectedTypes)) {  
 return isSameType(expectedTypes, type) ? 0 : -1;  
 }  
 return -1;  
}  
function validateProps(rawProps, props, instance) {  
 const resolvedValues = toRaw(props);  
 const options = instance.propsOptions[0];  
 for (const key in options) {  
 let opt = options[key];  
 if (opt == null) continue;  
 validateProp(  
 key,  
 resolvedValues[key],  
 opt,  
 true ? shallowReadonly(resolvedValues) : resolvedValues,  
 !hasOwn(rawProps, key) && !hasOwn(rawProps, hyphenate(key))  
 );  
 }  
}  
function validateProp(name, value, prop, props, isAbsent) {  
 const { type, required, validator, skipCheck } = prop;  
 if (required && isAbsent) {  
 warn$1('Missing required prop: "' + name + '"');  
 return;  
 }  
 if (value == null && !required) {  
 return;  
 }  
 if (type != null && type !== true && !skipCheck) {  
 let isValid = false;  
 const types = isArray(type) ? type : [type];  
 const expectedTypes = [];  
 for (let i = 0; i < types.length && !isValid; i++) {  
 const { valid, expectedType } = assertType(value, types[i]);  
 expectedTypes.push(expectedType || "");  
 isValid = valid;  
 }  
 if (!isValid) {  
 warn$1(getInvalidTypeMessage(name, value, expectedTypes));  
 return;  
 }  
 }  
 if (validator && !validator(value, props)) {  
 warn$1('Invalid prop: custom validator check failed for prop "' + name + '".');  
 }  
}  
var isSimpleType = makeMap(  
 "String,Number,Boolean,Function,Symbol,BigInt"  
);  
function assertType(value, type) {  
 let valid;  
 const expectedType = getType(type);  
 if (isSimpleType(expectedType)) {  
 const t = typeof value;  
 valid = t === expectedType.toLowerCase();  
 if (!valid && t === "object") {  
 valid = value instanceof type;  
 }  
 } else if (expectedType === "Object") {  
 valid = isObject(value);  
 } else if (expectedType === "Array") {  
 valid = isArray(value);  
 } else if (expectedType === "null") {  
 valid = value === null;  
 } else {  
 valid = value instanceof type;  
 }  
 return {  
 valid,  
 expectedType  
 };  
}  
function getInvalidTypeMessage(name, value, expectedTypes) {  
 if (expectedTypes.length === 0) {  
 return `Prop type [] for prop "${name}" won't match anything. Did you mean to use type Array instead?`;  
 }  
 let message = `Invalid prop: type check failed for prop "${name}". Expected ${expectedTypes.map(capitalize).join(" | ")}`;  
 const expectedType = expectedTypes[0];  
 const receivedType = toRawType(value);  
 const expectedValue = styleValue(value, expectedType);  
 const receivedValue = styleValue(value, receivedType);  
 if (expectedTypes.length === 1 && isExplicable(expectedType) && !isBoolean(expectedType, receivedType)) {  
 message += ` with value ${expectedValue}`;  
 }  
 message += `, got ${receivedType} `;  
 if (isExplicable(receivedType)) {  
 message += `with value ${receivedValue}.`;  
 }  
 return message;  
}  
function styleValue(value, type) {  
 if (type === "String") {  
 return `"${value}"`;  
 } else if (type === "Number") {  
 return `${Number(value)}`;  
 } else {  
 return `${value}`;  
 }  
}  
function isExplicable(type) {  
 const explicitTypes = ["string", "number", "boolean"];  
 return explicitTypes.some((elem) => type.toLowerCase() === elem);  
}  
function isBoolean(...args) {  
 return args.some((elem) => elem.toLowerCase() === "boolean");  
}  
var isInternalKey = (key) => key[0] === "\_" || key === "$stable";  
var normalizeSlotValue = (value) => isArray(value) ? value.map(normalizeVNode) : [normalizeVNode(value)];  
var normalizeSlot = (key, rawSlot, ctx) => {  
 if (rawSlot.\_n) {  
 return rawSlot;  
 }  
 const normalized = withCtx((...args) => {  
 if (currentInstance && (!ctx || ctx.root === currentInstance.root)) {  
 warn$1(  
 `Slot "${key}" invoked outside of the render function: this will not track dependencies used in the slot. Invoke the slot function inside the render function instead.`  
 );  
 }  
 return normalizeSlotValue(rawSlot(...args));  
 }, ctx);  
 normalized.\_c = false;  
 return normalized;  
};  
var normalizeObjectSlots = (rawSlots, slots, instance) => {  
 const ctx = rawSlots.\_ctx;  
 for (const key in rawSlots) {  
 if (isInternalKey(key)) continue;  
 const value = rawSlots[key];  
 if (isFunction(value)) {  
 slots[key] = normalizeSlot(key, value, ctx);  
 } else if (value != null) {  
 if (true) {  
 warn$1(  
 `Non-function value encountered for slot "${key}". Prefer function slots for better performance.`  
 );  
 }  
 const normalized = normalizeSlotValue(value);  
 slots[key] = () => normalized;  
 }  
 }  
};  
var normalizeVNodeSlots = (instance, children) => {  
 if (!isKeepAlive(instance.vnode) && true) {  
 warn$1(  
 `Non-function value encountered for default slot. Prefer function slots for better performance.`  
 );  
 }  
 const normalized = normalizeSlotValue(children);  
 instance.slots.default = () => normalized;  
};  
var assignSlots = (slots, children, optimized) => {  
 for (const key in children) {  
 if (optimized || key !== "\_") {  
 slots[key] = children[key];  
 }  
 }  
};  
var initSlots = (instance, children, optimized) => {  
 const slots = instance.slots = createInternalObject();  
 if (instance.vnode.shapeFlag & 32) {  
 const type = children.\_;  
 if (type) {  
 assignSlots(slots, children, optimized);  
 if (optimized) {  
 def(slots, "\_", type, true);  
 }  
 } else {  
 normalizeObjectSlots(children, slots);  
 }  
 } else if (children) {  
 normalizeVNodeSlots(instance, children);  
 }  
};  
var updateSlots = (instance, children, optimized) => {  
 const { vnode, slots } = instance;  
 let needDeletionCheck = true;  
 let deletionComparisonTarget = EMPTY\_OBJ;  
 if (vnode.shapeFlag & 32) {  
 const type = children.\_;  
 if (type) {  
 if (isHmrUpdating) {  
 assignSlots(slots, children, optimized);  
 trigger(instance, "set", "$slots");  
 } else if (optimized && type === 1) {  
 needDeletionCheck = false;  
 } else {  
 assignSlots(slots, children, optimized);  
 }  
 } else {  
 needDeletionCheck = !children.$stable;  
 normalizeObjectSlots(children, slots);  
 }  
 deletionComparisonTarget = children;  
 } else if (children) {  
 normalizeVNodeSlots(instance, children);  
 deletionComparisonTarget = { default: 1 };  
 }  
 if (needDeletionCheck) {  
 for (const key in slots) {  
 if (!isInternalKey(key) && deletionComparisonTarget[key] == null) {  
 delete slots[key];  
 }  
 }  
 }  
};  
function setRef(rawRef, oldRawRef, parentSuspense, vnode, isUnmount = false) {  
 if (isArray(rawRef)) {  
 rawRef.forEach(  
 (r, i) => setRef(  
 r,  
 oldRawRef && (isArray(oldRawRef) ? oldRawRef[i] : oldRawRef),  
 parentSuspense,  
 vnode,  
 isUnmount  
 )  
 );  
 return;  
 }  
 if (isAsyncWrapper(vnode) && !isUnmount) {  
 return;  
 }  
 const refValue = vnode.shapeFlag & 4 ? getComponentPublicInstance(vnode.component) : vnode.el;  
 const value = isUnmount ? null : refValue;  
 const { i: owner, r: ref2 } = rawRef;  
 if (!owner) {  
 warn$1(  
 `Missing ref owner context. ref cannot be used on hoisted vnodes. A vnode with ref must be created inside the render function.`  
 );  
 return;  
 }  
 const oldRef = oldRawRef && oldRawRef.r;  
 const refs = owner.refs === EMPTY\_OBJ ? owner.refs = {} : owner.refs;  
 const setupState = owner.setupState;  
 if (oldRef != null && oldRef !== ref2) {  
 if (isString(oldRef)) {  
 refs[oldRef] = null;  
 if (hasOwn(setupState, oldRef)) {  
 setupState[oldRef] = null;  
 }  
 } else if (isRef2(oldRef)) {  
 oldRef.value = null;  
 }  
 }  
 if (isFunction(ref2)) {  
 callWithErrorHandling(ref2, owner, 12, [value, refs]);  
 } else {  
 const \_isString = isString(ref2);  
 const \_isRef = isRef2(ref2);  
 if (\_isString || \_isRef) {  
 const doSet = () => {  
 if (rawRef.f) {  
 const existing = \_isString ? hasOwn(setupState, ref2) ? setupState[ref2] : refs[ref2] : ref2.value;  
 if (isUnmount) {  
 isArray(existing) && remove(existing, refValue);  
 } else {  
 if (!isArray(existing)) {  
 if (\_isString) {  
 refs[ref2] = [refValue];  
 if (hasOwn(setupState, ref2)) {  
 setupState[ref2] = refs[ref2];  
 }  
 } else {  
 ref2.value = [refValue];  
 if (rawRef.k) refs[rawRef.k] = ref2.value;  
 }  
 } else if (!existing.includes(refValue)) {  
 existing.push(refValue);  
 }  
 }  
 } else if (\_isString) {  
 refs[ref2] = value;  
 if (hasOwn(setupState, ref2)) {  
 setupState[ref2] = value;  
 }  
 } else if (\_isRef) {  
 ref2.value = value;  
 if (rawRef.k) refs[rawRef.k] = value;  
 } else if (true) {  
 warn$1("Invalid template ref type:", ref2, `(${typeof ref2})`);  
 }  
 };  
 if (value) {  
 doSet.id = -1;  
 queuePostRenderEffect(doSet, parentSuspense);  
 } else {  
 doSet();  
 }  
 } else if (true) {  
 warn$1("Invalid template ref type:", ref2, `(${typeof ref2})`);  
 }  
 }  
}  
var TeleportEndKey = Symbol("\_vte");  
var isTeleport = (type) => type.\_\_isTeleport;  
var isTeleportDisabled = (props) => props && (props.disabled || props.disabled === "");  
var isTargetSVG = (target) => typeof SVGElement !== "undefined" && target instanceof SVGElement;  
var isTargetMathML = (target) => typeof MathMLElement === "function" && target instanceof MathMLElement;  
var resolveTarget = (props, select) => {  
 const targetSelector = props && props.to;  
 if (isString(targetSelector)) {  
 if (!select) {  
 warn$1(  
 `Current renderer does not support string target for Teleports. (missing querySelector renderer option)`  
 );  
 return null;  
 } else {  
 const target = select(targetSelector);  
 if (!target && !isTeleportDisabled(props)) {  
 warn$1(  
 `Failed to locate Teleport target with selector "${targetSelector}". Note the target element must exist before the component is mounted - i.e. the target cannot be rendered by the component itself, and ideally should be outside of the entire Vue component tree.`  
 );  
 }  
 return target;  
 }  
 } else {  
 if (!targetSelector && !isTeleportDisabled(props)) {  
 warn$1(`Invalid Teleport target: ${targetSelector}`);  
 }  
 return targetSelector;  
 }  
};  
var TeleportImpl = {  
 name: "Teleport",  
 \_\_isTeleport: true,  
 process(n1, n2, container, anchor, parentComponent, parentSuspense, namespace, slotScopeIds, optimized, internals) {  
 const {  
 mc: mountChildren,  
 pc: patchChildren,  
 pbc: patchBlockChildren,  
 o: { insert, querySelector, createText, createComment }  
 } = internals;  
 const disabled = isTeleportDisabled(n2.props);  
 let { shapeFlag, children, dynamicChildren } = n2;  
 if (isHmrUpdating) {  
 optimized = false;  
 dynamicChildren = null;  
 }  
 if (n1 == null) {  
 const placeholder = n2.el = true ? createComment("teleport start") : createText("");  
 const mainAnchor = n2.anchor = true ? createComment("teleport end") : createText("");  
 const target = n2.target = resolveTarget(n2.props, querySelector);  
 const targetStart = n2.targetStart = createText("");  
 const targetAnchor = n2.targetAnchor = createText("");  
 insert(placeholder, container, anchor);  
 insert(mainAnchor, container, anchor);  
 targetStart[TeleportEndKey] = targetAnchor;  
 if (target) {  
 insert(targetStart, target);  
 insert(targetAnchor, target);  
 if (namespace === "svg" || isTargetSVG(target)) {  
 namespace = "svg";  
 } else if (namespace === "mathml" || isTargetMathML(target)) {  
 namespace = "mathml";  
 }  
 } else if (!disabled) {  
 warn$1("Invalid Teleport target on mount:", target, `(${typeof target})`);  
 }  
 const mount = (container2, anchor2) => {  
 if (shapeFlag & 16) {  
 mountChildren(  
 children,  
 container2,  
 anchor2,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 }  
 };  
 if (disabled) {  
 mount(container, mainAnchor);  
 } else if (target) {  
 mount(target, targetAnchor);  
 }  
 } else {  
 n2.el = n1.el;  
 n2.targetStart = n1.targetStart;  
 const mainAnchor = n2.anchor = n1.anchor;  
 const target = n2.target = n1.target;  
 const targetAnchor = n2.targetAnchor = n1.targetAnchor;  
 const wasDisabled = isTeleportDisabled(n1.props);  
 const currentContainer = wasDisabled ? container : target;  
 const currentAnchor = wasDisabled ? mainAnchor : targetAnchor;  
 if (namespace === "svg" || isTargetSVG(target)) {  
 namespace = "svg";  
 } else if (namespace === "mathml" || isTargetMathML(target)) {  
 namespace = "mathml";  
 }  
 if (dynamicChildren) {  
 patchBlockChildren(  
 n1.dynamicChildren,  
 dynamicChildren,  
 currentContainer,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds  
 );  
 traverseStaticChildren(n1, n2, true);  
 } else if (!optimized) {  
 patchChildren(  
 n1,  
 n2,  
 currentContainer,  
 currentAnchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 false  
 );  
 }  
 if (disabled) {  
 if (!wasDisabled) {  
 moveTeleport(  
 n2,  
 container,  
 mainAnchor,  
 internals,  
 1  
 );  
 } else {  
 if (n2.props && n1.props && n2.props.to !== n1.props.to) {  
 n2.props.to = n1.props.to;  
 }  
 }  
 } else {  
 if ((n2.props && n2.props.to) !== (n1.props && n1.props.to)) {  
 const nextTarget = n2.target = resolveTarget(  
 n2.props,  
 querySelector  
 );  
 if (nextTarget) {  
 moveTeleport(  
 n2,  
 nextTarget,  
 null,  
 internals,  
 0  
 );  
 } else if (true) {  
 warn$1(  
 "Invalid Teleport target on update:",  
 target,  
 `(${typeof target})`  
 );  
 }  
 } else if (wasDisabled) {  
 moveTeleport(  
 n2,  
 target,  
 targetAnchor,  
 internals,  
 1  
 );  
 }  
 }  
 }  
 updateCssVars(n2);  
 },  
 remove(vnode, parentComponent, parentSuspense, { um: unmount, o: { remove: hostRemove } }, doRemove) {  
 const {  
 shapeFlag,  
 children,  
 anchor,  
 targetStart,  
 targetAnchor,  
 target,  
 props  
 } = vnode;  
 if (target) {  
 hostRemove(targetStart);  
 hostRemove(targetAnchor);  
 }  
 doRemove && hostRemove(anchor);  
 if (shapeFlag & 16) {  
 const shouldRemove = doRemove || !isTeleportDisabled(props);  
 for (let i = 0; i < children.length; i++) {  
 const child = children[i];  
 unmount(  
 child,  
 parentComponent,  
 parentSuspense,  
 shouldRemove,  
 !!child.dynamicChildren  
 );  
 }  
 }  
 },  
 move: moveTeleport,  
 hydrate: hydrateTeleport  
};  
function moveTeleport(vnode, container, parentAnchor, { o: { insert }, m: move }, moveType = 2) {  
 if (moveType === 0) {  
 insert(vnode.targetAnchor, container, parentAnchor);  
 }  
 const { el, anchor, shapeFlag, children, props } = vnode;  
 const isReorder = moveType === 2;  
 if (isReorder) {  
 insert(el, container, parentAnchor);  
 }  
 if (!isReorder || isTeleportDisabled(props)) {  
 if (shapeFlag & 16) {  
 for (let i = 0; i < children.length; i++) {  
 move(  
 children[i],  
 container,  
 parentAnchor,  
 2  
 );  
 }  
 }  
 }  
 if (isReorder) {  
 insert(anchor, container, parentAnchor);  
 }  
}  
function hydrateTeleport(node, vnode, parentComponent, parentSuspense, slotScopeIds, optimized, {  
 o: { nextSibling, parentNode, querySelector }  
}, hydrateChildren) {  
 const target = vnode.target = resolveTarget(  
 vnode.props,  
 querySelector  
 );  
 if (target) {  
 const targetNode = target.\_lpa || target.firstChild;  
 if (vnode.shapeFlag & 16) {  
 if (isTeleportDisabled(vnode.props)) {  
 vnode.anchor = hydrateChildren(  
 nextSibling(node),  
 vnode,  
 parentNode(node),  
 parentComponent,  
 parentSuspense,  
 slotScopeIds,  
 optimized  
 );  
 vnode.targetAnchor = targetNode;  
 } else {  
 vnode.anchor = nextSibling(node);  
 let targetAnchor = targetNode;  
 while (targetAnchor) {  
 targetAnchor = nextSibling(targetAnchor);  
 if (targetAnchor && targetAnchor.nodeType === 8 && targetAnchor.data === "teleport anchor") {  
 vnode.targetAnchor = targetAnchor;  
 target.\_lpa = vnode.targetAnchor && nextSibling(vnode.targetAnchor);  
 break;  
 }  
 }  
 hydrateChildren(  
 targetNode,  
 vnode,  
 target,  
 parentComponent,  
 parentSuspense,  
 slotScopeIds,  
 optimized  
 );  
 }  
 }  
 updateCssVars(vnode);  
 }  
 return vnode.anchor && nextSibling(vnode.anchor);  
}  
var Teleport = TeleportImpl;  
function updateCssVars(vnode) {  
 const ctx = vnode.ctx;  
 if (ctx && ctx.ut) {  
 let node = vnode.children[0].el;  
 while (node && node !== vnode.targetAnchor) {  
 if (node.nodeType === 1) node.setAttribute("data-v-owner", ctx.uid);  
 node = node.nextSibling;  
 }  
 ctx.ut();  
 }  
}  
var hasLoggedMismatchError = false;  
var logMismatchError = () => {  
 if (hasLoggedMismatchError) {  
 return;  
 }  
 console.error("Hydration completed but contains mismatches.");  
 hasLoggedMismatchError = true;  
};  
var isSVGContainer = (container) => container.namespaceURI.includes("svg") && container.tagName !== "foreignObject";  
var isMathMLContainer = (container) => container.namespaceURI.includes("MathML");  
var getContainerType = (container) => {  
 if (isSVGContainer(container)) return "svg";  
 if (isMathMLContainer(container)) return "mathml";  
 return void 0;  
};  
var isComment = (node) => node.nodeType === 8;  
function createHydrationFunctions(rendererInternals) {  
 const {  
 mt: mountComponent,  
 p: patch,  
 o: {  
 patchProp: patchProp2,  
 createText,  
 nextSibling,  
 parentNode,  
 remove: remove2,  
 insert,  
 createComment  
 }  
 } = rendererInternals;  
 const hydrate2 = (vnode, container) => {  
 if (!container.hasChildNodes()) {  
 warn$1(  
 `Attempting to hydrate existing markup but container is empty. Performing full mount instead.`  
 );  
 patch(null, vnode, container);  
 flushPostFlushCbs();  
 container.\_vnode = vnode;  
 return;  
 }  
 hydrateNode(container.firstChild, vnode, null, null, null);  
 flushPostFlushCbs();  
 container.\_vnode = vnode;  
 };  
 const hydrateNode = (node, vnode, parentComponent, parentSuspense, slotScopeIds, optimized = false) => {  
 optimized = optimized || !!vnode.dynamicChildren;  
 const isFragmentStart = isComment(node) && node.data === "[";  
 const onMismatch = () => handleMismatch(  
 node,  
 vnode,  
 parentComponent,  
 parentSuspense,  
 slotScopeIds,  
 isFragmentStart  
 );  
 const { type, ref: ref2, shapeFlag, patchFlag } = vnode;  
 let domType = node.nodeType;  
 vnode.el = node;  
 if (true) {  
 def(node, "\_\_vnode", vnode, true);  
 def(node, "\_\_vueParentComponent", parentComponent, true);  
 }  
 if (patchFlag === -2) {  
 optimized = false;  
 vnode.dynamicChildren = null;  
 }  
 let nextNode = null;  
 switch (type) {  
 case Text:  
 if (domType !== 3) {  
 if (vnode.children === "") {  
 insert(vnode.el = createText(""), parentNode(node), node);  
 nextNode = node;  
 } else {  
 nextNode = onMismatch();  
 }  
 } else {  
 if (node.data !== vnode.children) {  
 warn$1(  
 `Hydration text mismatch in`,  
 node.parentNode,  
 `  
 - rendered on server: ${JSON.stringify(  
 node.data  
 )}  
 - expected on client: ${JSON.stringify(vnode.children)}`  
 );  
 logMismatchError();  
 node.data = vnode.children;  
 }  
 nextNode = nextSibling(node);  
 }  
 break;  
 case Comment:  
 if (isTemplateNode(node)) {  
 nextNode = nextSibling(node);  
 replaceNode(  
 vnode.el = node.content.firstChild,  
 node,  
 parentComponent  
 );  
 } else if (domType !== 8 || isFragmentStart) {  
 nextNode = onMismatch();  
 } else {  
 nextNode = nextSibling(node);  
 }  
 break;  
 case Static:  
 if (isFragmentStart) {  
 node = nextSibling(node);  
 domType = node.nodeType;  
 }  
 if (domType === 1 || domType === 3) {  
 nextNode = node;  
 const needToAdoptContent = !vnode.children.length;  
 for (let i = 0; i < vnode.staticCount; i++) {  
 if (needToAdoptContent)  
 vnode.children += nextNode.nodeType === 1 ? nextNode.outerHTML : nextNode.data;  
 if (i === vnode.staticCount - 1) {  
 vnode.anchor = nextNode;  
 }  
 nextNode = nextSibling(nextNode);  
 }  
 return isFragmentStart ? nextSibling(nextNode) : nextNode;  
 } else {  
 onMismatch();  
 }  
 break;  
 case Fragment:  
 if (!isFragmentStart) {  
 nextNode = onMismatch();  
 } else {  
 nextNode = hydrateFragment(  
 node,  
 vnode,  
 parentComponent,  
 parentSuspense,  
 slotScopeIds,  
 optimized  
 );  
 }  
 break;  
 default:  
 if (shapeFlag & 1) {  
 if ((domType !== 1 || vnode.type.toLowerCase() !== node.tagName.toLowerCase()) && !isTemplateNode(node)) {  
 nextNode = onMismatch();  
 } else {  
 nextNode = hydrateElement(  
 node,  
 vnode,  
 parentComponent,  
 parentSuspense,  
 slotScopeIds,  
 optimized  
 );  
 }  
 } else if (shapeFlag & 6) {  
 vnode.slotScopeIds = slotScopeIds;  
 const container = parentNode(node);  
 if (isFragmentStart) {  
 nextNode = locateClosingAnchor(node);  
 } else if (isComment(node) && node.data === "teleport start") {  
 nextNode = locateClosingAnchor(node, node.data, "teleport end");  
 } else {  
 nextNode = nextSibling(node);  
 }  
 mountComponent(  
 vnode,  
 container,  
 null,  
 parentComponent,  
 parentSuspense,  
 getContainerType(container),  
 optimized  
 );  
 if (isAsyncWrapper(vnode)) {  
 let subTree;  
 if (isFragmentStart) {  
 subTree = createVNode(Fragment);  
 subTree.anchor = nextNode ? nextNode.previousSibling : container.lastChild;  
 } else {  
 subTree = node.nodeType === 3 ? createTextVNode("") : createVNode("div");  
 }  
 subTree.el = node;  
 vnode.component.subTree = subTree;  
 }  
 } else if (shapeFlag & 64) {  
 if (domType !== 8) {  
 nextNode = onMismatch();  
 } else {  
 nextNode = vnode.type.hydrate(  
 node,  
 vnode,  
 parentComponent,  
 parentSuspense,  
 slotScopeIds,  
 optimized,  
 rendererInternals,  
 hydrateChildren  
 );  
 }  
 } else if (shapeFlag & 128) {  
 nextNode = vnode.type.hydrate(  
 node,  
 vnode,  
 parentComponent,  
 parentSuspense,  
 getContainerType(parentNode(node)),  
 slotScopeIds,  
 optimized,  
 rendererInternals,  
 hydrateNode  
 );  
 } else if (true) {  
 warn$1("Invalid HostVNode type:", type, `(${typeof type})`);  
 }  
 }  
 if (ref2 != null) {  
 setRef(ref2, null, parentSuspense, vnode);  
 }  
 return nextNode;  
 };  
 const hydrateElement = (el, vnode, parentComponent, parentSuspense, slotScopeIds, optimized) => {  
 optimized = optimized || !!vnode.dynamicChildren;  
 const { type, props, patchFlag, shapeFlag, dirs, transition } = vnode;  
 const forcePatch = type === "input" || type === "option";  
 if (true) {  
 if (dirs) {  
 invokeDirectiveHook(vnode, null, parentComponent, "created");  
 }  
 let needCallTransitionHooks = false;  
 if (isTemplateNode(el)) {  
  
 const content = el.content.firstChild;  
 if (needCallTransitionHooks) {  
 transition.beforeEnter(content);  
 }  
 replaceNode(content, el, parentComponent);  
 vnode.el = el = content;  
 }  
 if (shapeFlag & 16 && // skip if element has innerHTML / textContent  
 !(props && (props.innerHTML || props.textContent))) {  
 let next = hydrateChildren(  
 el.firstChild,  
 vnode,  
 el,  
 parentComponent,  
 parentSuspense,  
 slotScopeIds,  
 optimized  
 );  
 let hasWarned2 = false;  
 while (next) {  
 if (!hasWarned2) {  
 warn$1(  
 `Hydration children mismatch on`,  
 el,  
 `  
Server rendered element contains more child nodes than client vdom.`  
 );  
 hasWarned2 = true;  
 }  
 logMismatchError();  
 const cur = next;  
 next = next.nextSibling;  
 remove2(cur);  
 }  
 } else if (shapeFlag & 8) {  
 if (el.textContent !== vnode.children) {  
 warn$1(  
 `Hydration text content mismatch on`,  
 el,  
 `  
 - rendered on server: ${el.textContent}  
 - expected on client: ${vnode.children}`  
 );  
 logMismatchError();  
 el.textContent = vnode.children;  
 }  
 }  
 if (props) {  
 if (true) {  
 for (const key in props) {  
 if (// #11189 skip if this node has directives that have created hooks  
 !(dirs && dirs.some((d) => d.dir.created)) && propHasMismatch(el, key, props[key], vnode, parentComponent)) {  
 logMismatchError();  
 }  
 if (forcePatch && (key.endsWith("value") || key === "indeterminate") || isOn(key) && !isReservedProp(key) || // force hydrate v-bind with .prop modifiers  
 key[0] === ".") {  
 patchProp2(el, key, null, props[key], void 0, parentComponent);  
 }  
 }  
 } else if (props.onClick) {  
 patchProp2(  
 el,  
 "onClick",  
 null,  
 props.onClick,  
 void 0,  
 parentComponent  
 );  
 } else if (patchFlag & 4 && isReactive(props.style)) {  
 for (const key in props.style) props.style[key];  
 }  
 }  
 let vnodeHooks;  
 if (vnodeHooks = props && props.onVnodeBeforeMount) {  
 invokeVNodeHook(vnodeHooks, parentComponent, vnode);  
 }  
 if (dirs) {  
 invokeDirectiveHook(vnode, null, parentComponent, "beforeMount");  
 }  
 if ((vnodeHooks = props && props.onVnodeMounted) || dirs || needCallTransitionHooks) {  
 queueEffectWithSuspense(() => {  
 vnodeHooks && invokeVNodeHook(vnodeHooks, parentComponent, vnode);  
 needCallTransitionHooks && transition.enter(el);  
 dirs && invokeDirectiveHook(vnode, null, parentComponent, "mounted");  
 }, parentSuspense);  
 }  
 }  
 return el.nextSibling;  
 };  
 const hydrateChildren = (node, parentVNode, container, parentComponent, parentSuspense, slotScopeIds, optimized) => {  
 optimized = optimized || !!parentVNode.dynamicChildren;  
 const children = parentVNode.children;  
 const l = children.length;  
 let hasWarned2 = false;  
 for (let i = 0; i < l; i++) {  
 const vnode = optimized ? children[i] : children[i] = normalizeVNode(children[i]);  
 const isText = vnode.type === Text;  
 if (node) {  
 if (isText && !optimized) {  
 let next = children[i + 1];  
 if (next && (next = normalizeVNode(next)).type === Text) {  
 insert(  
 createText(  
 node.data.slice(vnode.children.length)  
 ),  
 container,  
 nextSibling(node)  
 );  
 node.data = vnode.children;  
 }  
 }  
 node = hydrateNode(  
 node,  
 vnode,  
 parentComponent,  
 parentSuspense,  
 slotScopeIds,  
 optimized  
 );  
 } else if (isText && !vnode.children) {  
 insert(vnode.el = createText(""), container);  
 } else {  
 if (!hasWarned2) {  
 warn$1(  
 `Hydration children mismatch on`,  
 container,  
 `  
Server rendered element contains fewer child nodes than client vdom.`  
 );  
 hasWarned2 = true;  
 }  
 logMismatchError();  
 patch(  
 null,  
 vnode,  
 container,  
 null,  
 parentComponent,  
 parentSuspense,  
 getContainerType(container),  
 slotScopeIds  
 );  
 }  
 }  
 return node;  
 };  
 const hydrateFragment = (node, vnode, parentComponent, parentSuspense, slotScopeIds, optimized) => {  
 const { slotScopeIds: fragmentSlotScopeIds } = vnode;  
 if (fragmentSlotScopeIds) {  
 slotScopeIds = slotScopeIds ? slotScopeIds.concat(fragmentSlotScopeIds) : fragmentSlotScopeIds;  
 }  
 const container = parentNode(node);  
 const next = hydrateChildren(  
 nextSibling(node),  
 vnode,  
 container,  
 parentComponent,  
 parentSuspense,  
 slotScopeIds,  
 optimized  
 );  
 if (next && isComment(next) && next.data === "]") {  
 return nextSibling(vnode.anchor = next);  
 } else {  
 logMismatchError();  
 insert(vnode.anchor = createComment(`]`), container, next);  
 return next;  
 }  
 };  
 const handleMismatch = (node, vnode, parentComponent, parentSuspense, slotScopeIds, isFragment) => {  
 warn$1(  
 `Hydration node mismatch:  
- rendered on server:`,  
 node,  
 node.nodeType === 3 ? `(text)` : isComment(node) && node.data === "[" ? `(start of fragment)` : ``,  
 `  
- expected on client:`,  
 vnode.type  
 );  
 logMismatchError();  
 vnode.el = null;  
 if (isFragment) {  
 const end = locateClosingAnchor(node);  
 while (true) {  
 const next2 = nextSibling(node);  
 if (next2 && next2 !== end) {  
 remove2(next2);  
 } else {  
 break;  
 }  
 }  
 }  
 const next = nextSibling(node);  
 const container = parentNode(node);  
 remove2(node);  
 patch(  
 null,  
 vnode,  
 container,  
 next,  
 parentComponent,  
 parentSuspense,  
 getContainerType(container),  
 slotScopeIds  
 );  
 return next;  
 };  
 const locateClosingAnchor = (node, open = "[", close = "]") => {  
 let match = 0;  
 while (node) {  
 node = nextSibling(node);  
 if (node && isComment(node)) {  
 if (node.data === open) match++;  
 if (node.data === close) {  
 if (match === 0) {  
 return nextSibling(node);  
 } else {  
 match--;  
 }  
 }  
 }  
 }  
 return node;  
 };  
 const replaceNode = (newNode, oldNode, parentComponent) => {  
 const parentNode2 = oldNode.parentNode;  
 if (parentNode2) {  
 parentNode2.replaceChild(newNode, oldNode);  
 }  
 let parent = parentComponent;  
 while (parent) {  
 if (parent.vnode.el === oldNode) {  
 parent.vnode.el = parent.subTree.el = newNode;  
 }  
 parent = parent.parent;  
 }  
 };  
 const isTemplateNode = (node) => {  
 return node.nodeType === 1 && node.tagName.toLowerCase() === "template";  
 };  
 return [hydrate2, hydrateNode];  
}  
function propHasMismatch(el, key, clientValue, vnode, instance) {  
 let mismatchType;  
 let mismatchKey;  
 let actual;  
 let expected;  
 if (key === "class") {  
 actual = el.getAttribute("class");  
 expected = normalizeClass(clientValue);  
 if (!isSetEqual(toClassSet(actual || ""), toClassSet(expected))) {  
 mismatchType = mismatchKey = `class`;  
 }  
 } else if (key === "style") {  
 actual = el.getAttribute("style") || "";  
 expected = isString(clientValue) ? clientValue : stringifyStyle(normalizeStyle(clientValue));  
 const actualMap = toStyleMap(actual);  
 const expectedMap = toStyleMap(expected);  
 if (vnode.dirs) {  
 for (const { dir, value } of vnode.dirs) {  
 if (dir.name === "show" && !value) {  
 expectedMap.set("display", "none");  
 }  
 }  
 }  
 if (instance) {  
 resolveCssVars(instance, vnode, expectedMap);  
 }  
 if (!isMapEqual(actualMap, expectedMap)) {  
 mismatchType = mismatchKey = "style";  
 }  
 } else if (el instanceof SVGElement && isKnownSvgAttr(key) || el instanceof HTMLElement && (isBooleanAttr(key) || isKnownHtmlAttr(key))) {  
 if (isBooleanAttr(key)) {  
 actual = el.hasAttribute(key);  
 expected = includeBooleanAttr(clientValue);  
 } else if (clientValue == null) {  
 actual = el.hasAttribute(key);  
 expected = false;  
 } else {  
 if (el.hasAttribute(key)) {  
 actual = el.getAttribute(key);  
 } else if (key === "value" && el.tagName === "TEXTAREA") {  
 actual = el.value;  
 } else {  
 actual = false;  
 }  
 expected = isRenderableAttrValue(clientValue) ? String(clientValue) : false;  
 }  
 if (actual !== expected) {  
 mismatchType = `attribute`;  
 mismatchKey = key;  
 }  
 }  
 if (mismatchType) {  
 const format = (v) => v === false ? `(not rendered)` : `${mismatchKey}="${v}"`;  
 const preSegment = `Hydration ${mismatchType} mismatch on`;  
 const postSegment = `  
 - rendered on server: ${format(actual)}  
 - expected on client: ${format(expected)}  
 Note: this mismatch is check-only. The DOM will not be rectified in production due to performance overhead.  
 You should fix the source of the mismatch.`;  
 {  
 warn$1(preSegment, el, postSegment);  
 }  
 return true;  
 }  
 return false;  
}  
function toClassSet(str) {  
 return new Set(str.trim().split(/\s+/));  
}  
function isSetEqual(a, b) {  
 if (a.size !== b.size) {  
 return false;  
 }  
 for (const s of a) {  
 if (!b.has(s)) {  
 return false;  
 }  
 }  
 return true;  
}  
function toStyleMap(str) {  
 const styleMap = /\* @\_\_PURE\_\_ \*/ new Map();  
 for (const item of str.split(";")) {  
 let [key, value] = item.split(":");  
 key = key.trim();  
 value = value && value.trim();  
 if (key && value) {  
 styleMap.set(key, value);  
 }  
 }  
 return styleMap;  
}  
function isMapEqual(a, b) {  
 if (a.size !== b.size) {  
 return false;  
 }  
 for (const [key, value] of a) {  
 if (value !== b.get(key)) {  
 return false;  
 }  
 }  
 return true;  
}  
function resolveCssVars(instance, vnode, expectedMap) {  
 const root = instance.subTree;  
  
 const cssVars = instance.getCssVars();  
 for (const key in cssVars) {  
 expectedMap.set(`--${key}`, String(cssVars[key]));  
 }  
 }  
 if (vnode === root && instance.parent) {  
 resolveCssVars(instance.parent, instance.vnode, expectedMap);  
 }  
}  
var supported;  
var perf;  
function startMeasure(instance, type) {  
 if (instance.appContext.config.performance && isSupported()) {  
 perf.mark(`vue-${type}-${instance.uid}`);  
 }  
 if (true) {  
 devtoolsPerfStart(instance, type, isSupported() ? perf.now() : Date.now());  
 }  
}  
function endMeasure(instance, type) {  
 if (instance.appContext.config.performance && isSupported()) {  
 const startTag = `vue-${type}-${instance.uid}`;  
 const endTag = startTag + `:end`;  
 perf.mark(endTag);  
 perf.measure(  
 `<${formatComponentName(instance, instance.type)}> ${type}`,  
 startTag,  
 endTag  
 );  
 perf.clearMarks(startTag);  
 perf.clearMarks(endTag);  
 }  
 if (true) {  
 devtoolsPerfEnd(instance, type, isSupported() ? perf.now() : Date.now());  
 }  
}  
function isSupported() {  
 if (supported !== void 0) {  
 return supported;  
 }  
 if (typeof window !== "undefined" && window.performance) {  
 supported = true;  
 perf = window.performance;  
 } else {  
 supported = false;  
 }  
 return supported;  
}  
function initFeatureFlags() {  
 const needWarn = [];  
 if (typeof \_\_VUE\_OPTIONS\_API\_\_ !== "boolean") {  
 needWarn.push(`\_\_VUE\_OPTIONS\_API\_\_`);  
 getGlobalThis().\_\_VUE\_OPTIONS\_API\_\_ = true;  
 }  
 if (typeof \_\_VUE\_PROD\_DEVTOOLS\_\_ !== "boolean") {  
 needWarn.push(`\_\_VUE\_PROD\_DEVTOOLS\_\_`);  
 getGlobalThis().\_\_VUE\_PROD\_DEVTOOLS\_\_ = false;  
 }  
 if (typeof \_\_VUE\_PROD\_HYDRATION\_MISMATCH\_DETAILS\_\_ !== "boolean") {  
 needWarn.push(`\_\_VUE\_PROD\_HYDRATION\_MISMATCH\_DETAILS\_\_`);  
 getGlobalThis().\_\_VUE\_PROD\_HYDRATION\_MISMATCH\_DETAILS\_\_ = false;  
 }  
 if (needWarn.length) {  
 const multi = needWarn.length > 1;  
 console.warn(  
 `Feature flag${multi ? `s` : ``} ${needWarn.join(", ")} ${multi ? `are` : `is`} not explicitly defined. You are running the esm-bundler build of Vue, which expects these compile-time feature flags to be globally injected via the bundler config in order to get better tree-shaking in the production bundle.  
For more details, see https://link.vuejs.org/feature-flags.`  
 );  
 }  
}  
var queuePostRenderEffect = queueEffectWithSuspense;  
function createRenderer(options) {  
 return baseCreateRenderer(options);  
}  
function createHydrationRenderer(options) {  
 return baseCreateRenderer(options, createHydrationFunctions);  
}  
function baseCreateRenderer(options, createHydrationFns) {  
 {  
 initFeatureFlags();  
 }  
 const target = getGlobalThis();  
 target.\_\_VUE\_\_ = true;  
 if (true) {  
 setDevtoolsHook$1(target.\_\_VUE\_DEVTOOLS\_GLOBAL\_HOOK\_\_, target);  
 }  
 const {  
 insert: hostInsert,  
 remove: hostRemove,  
 patchProp: hostPatchProp,  
 createElement: hostCreateElement,  
 createText: hostCreateText,  
 createComment: hostCreateComment,  
 setText: hostSetText,  
 setElementText: hostSetElementText,  
 parentNode: hostParentNode,  
 nextSibling: hostNextSibling,  
 setScopeId: hostSetScopeId = NOOP,  
 insertStaticContent: hostInsertStaticContent  
 } = options;  
 const patch = (n1, n2, container, anchor = null, parentComponent = null, parentSuspense = null, namespace = void 0, slotScopeIds = null, optimized = isHmrUpdating ? false : !!n2.dynamicChildren) => {  
 if (n1 === n2) {  
 return;  
 }  
 if (n1 && !isSameVNodeType(n1, n2)) {  
 anchor = getNextHostNode(n1);  
 unmount(n1, parentComponent, parentSuspense, true);  
 n1 = null;  
 }  
 if (n2.patchFlag === -2) {  
 optimized = false;  
 n2.dynamicChildren = null;  
 }  
 const { type, ref: ref2, shapeFlag } = n2;  
 switch (type) {  
 case Text:  
 processText(n1, n2, container, anchor);  
 break;  
 case Comment:  
 processCommentNode(n1, n2, container, anchor);  
 break;  
 case Static:  
 if (n1 == null) {  
 mountStaticNode(n2, container, anchor, namespace);  
 } else if (true) {  
 patchStaticNode(n1, n2, container, namespace);  
 }  
 break;  
 case Fragment:  
 processFragment(  
 n1,  
 n2,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 break;  
 default:  
 if (shapeFlag & 1) {  
 processElement(  
 n1,  
 n2,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 } else if (shapeFlag & 6) {  
 processComponent(  
 n1,  
 n2,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 } else if (shapeFlag & 64) {  
 type.process(  
 n1,  
 n2,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized,  
 internals  
 );  
 } else if (shapeFlag & 128) {  
 type.process(  
 n1,  
 n2,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized,  
 internals  
 );  
 } else if (true) {  
 warn$1("Invalid VNode type:", type, `(${typeof type})`);  
 }  
 }  
 if (ref2 != null && parentComponent) {  
 setRef(ref2, n1 && n1.ref, parentSuspense, n2 || n1, !n2);  
 }  
 };  
 const processText = (n1, n2, container, anchor) => {  
 if (n1 == null) {  
 hostInsert(  
 n2.el = hostCreateText(n2.children),  
 container,  
 anchor  
 );  
 } else {  
 const el = n2.el = n1.el;  
 if (n2.children !== n1.children) {  
 hostSetText(el, n2.children);  
 }  
 }  
 };  
 const processCommentNode = (n1, n2, container, anchor) => {  
 if (n1 == null) {  
 hostInsert(  
 n2.el = hostCreateComment(n2.children || ""),  
 container,  
 anchor  
 );  
 } else {  
 n2.el = n1.el;  
 }  
 };  
 const mountStaticNode = (n2, container, anchor, namespace) => {  
 [n2.el, n2.anchor] = hostInsertStaticContent(  
 n2.children,  
 container,  
 anchor,  
 namespace,  
 n2.el,  
 n2.anchor  
 );  
 };  
 const patchStaticNode = (n1, n2, container, namespace) => {  
 if (n2.children !== n1.children) {  
 const anchor = hostNextSibling(n1.anchor);  
 removeStaticNode(n1);  
 [n2.el, n2.anchor] = hostInsertStaticContent(  
 n2.children,  
 container,  
 anchor,  
 namespace  
 );  
 } else {  
 n2.el = n1.el;  
 n2.anchor = n1.anchor;  
 }  
 };  
 const moveStaticNode = ({ el, anchor }, container, nextSibling) => {  
 let next;  
 while (el && el !== anchor) {  
 next = hostNextSibling(el);  
 hostInsert(el, container, nextSibling);  
 el = next;  
 }  
 hostInsert(anchor, container, nextSibling);  
 };  
 const removeStaticNode = ({ el, anchor }) => {  
 let next;  
 while (el && el !== anchor) {  
 next = hostNextSibling(el);  
 hostRemove(el);  
 el = next;  
 }  
 hostRemove(anchor);  
 };  
 const processElement = (n1, n2, container, anchor, parentComponent, parentSuspense, namespace, slotScopeIds, optimized) => {  
 if (n2.type === "svg") {  
 namespace = "svg";  
 } else if (n2.type === "math") {  
 namespace = "mathml";  
 }  
 if (n1 == null) {  
 mountElement(  
 n2,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 } else {  
 patchElement(  
 n1,  
 n2,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 }  
 };  
 const mountElement = (vnode, container, anchor, parentComponent, parentSuspense, namespace, slotScopeIds, optimized) => {  
 let el;  
 let vnodeHook;  
 const { props, shapeFlag, transition, dirs } = vnode;  
 el = vnode.el = hostCreateElement(  
 vnode.type,  
 namespace,  
 props && props.is,  
 props  
 );  
 if (shapeFlag & 8) {  
 hostSetElementText(el, vnode.children);  
 } else if (shapeFlag & 16) {  
 mountChildren(  
 vnode.children,  
 el,  
 null,  
 parentComponent,  
 parentSuspense,  
 resolveChildrenNamespace(vnode, namespace),  
 slotScopeIds,  
 optimized  
 );  
 }  
 if (dirs) {  
 invokeDirectiveHook(vnode, null, parentComponent, "created");  
 }  
 setScopeId(el, vnode, vnode.scopeId, slotScopeIds, parentComponent);  
 if (props) {  
 for (const key in props) {  
 if (key !== "value" && !isReservedProp(key)) {  
 hostPatchProp(el, key, null, props[key], namespace, parentComponent);  
 }  
 }  
 if ("value" in props) {  
 hostPatchProp(el, "value", null, props.value, namespace);  
 }  
 if (vnodeHook = props.onVnodeBeforeMount) {  
 invokeVNodeHook(vnodeHook, parentComponent, vnode);  
 }  
 }  
 if (true) {  
 def(el, "\_\_vnode", vnode, true);  
 def(el, "\_\_vueParentComponent", parentComponent, true);  
 }  
 if (dirs) {  
 invokeDirectiveHook(vnode, null, parentComponent, "beforeMount");  
 }  
 const needCallTransitionHooks = needTransition(parentSuspense, transition);  
 if (needCallTransitionHooks) {  
 transition.beforeEnter(el);  
 }  
 hostInsert(el, container, anchor);  
 if ((vnodeHook = props && props.onVnodeMounted) || needCallTransitionHooks || dirs) {  
 queuePostRenderEffect(() => {  
 vnodeHook && invokeVNodeHook(vnodeHook, parentComponent, vnode);  
 needCallTransitionHooks && transition.enter(el);  
 dirs && invokeDirectiveHook(vnode, null, parentComponent, "mounted");  
 }, parentSuspense);  
 }  
 };  
 const setScopeId = (el, vnode, scopeId, slotScopeIds, parentComponent) => {  
 if (scopeId) {  
 hostSetScopeId(el, scopeId);  
 }  
 if (slotScopeIds) {  
 for (let i = 0; i < slotScopeIds.length; i++) {  
 hostSetScopeId(el, slotScopeIds[i]);  
 }  
 }  
 if (parentComponent) {  
 let subTree = parentComponent.subTree;  
 if (subTree.patchFlag > 0 && subTree.patchFlag & 2048) {  
 subTree = filterSingleRoot(subTree.children) || subTree;  
 }  
 if (vnode === subTree) {  
 const parentVNode = parentComponent.vnode;  
 setScopeId(  
 el,  
 parentVNode,  
 parentVNode.scopeId,  
 parentVNode.slotScopeIds,  
 parentComponent.parent  
 );  
 }  
 }  
 };  
 const mountChildren = (children, container, anchor, parentComponent, parentSuspense, namespace, slotScopeIds, optimized, start = 0) => {  
 for (let i = start; i < children.length; i++) {  
 const child = children[i] = optimized ? cloneIfMounted(children[i]) : normalizeVNode(children[i]);  
 patch(  
 null,  
 child,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 }  
 };  
 const patchElement = (n1, n2, parentComponent, parentSuspense, namespace, slotScopeIds, optimized) => {  
 const el = n2.el = n1.el;  
 if (true) {  
 el.\_\_vnode = n2;  
 }  
 let { patchFlag, dynamicChildren, dirs } = n2;  
 patchFlag |= n1.patchFlag & 16;  
 const oldProps = n1.props || EMPTY\_OBJ;  
 const newProps = n2.props || EMPTY\_OBJ;  
 let vnodeHook;  
 parentComponent && toggleRecurse(parentComponent, false);  
 if (vnodeHook = newProps.onVnodeBeforeUpdate) {  
 invokeVNodeHook(vnodeHook, parentComponent, n2, n1);  
 }  
 if (dirs) {  
 invokeDirectiveHook(n2, n1, parentComponent, "beforeUpdate");  
 }  
 parentComponent && toggleRecurse(parentComponent, true);  
 if (isHmrUpdating) {  
 patchFlag = 0;  
 optimized = false;  
 dynamicChildren = null;  
 }  
 if (oldProps.innerHTML && newProps.innerHTML == null || oldProps.textContent && newProps.textContent == null) {  
 hostSetElementText(el, "");  
 }  
 if (dynamicChildren) {  
 patchBlockChildren(  
 n1.dynamicChildren,  
 dynamicChildren,  
 el,  
 parentComponent,  
 parentSuspense,  
 resolveChildrenNamespace(n2, namespace),  
 slotScopeIds  
 );  
 if (true) {  
 traverseStaticChildren(n1, n2);  
 }  
 } else if (!optimized) {  
 patchChildren(  
 n1,  
 n2,  
 el,  
 null,  
 parentComponent,  
 parentSuspense,  
 resolveChildrenNamespace(n2, namespace),  
 slotScopeIds,  
 false  
 );  
 }  
 if (patchFlag > 0) {  
 if (patchFlag & 16) {  
 patchProps(el, oldProps, newProps, parentComponent, namespace);  
 } else {  
 if (patchFlag & 2) {  
 if (oldProps.class !== newProps.class) {  
 hostPatchProp(el, "class", null, newProps.class, namespace);  
 }  
 }  
 if (patchFlag & 4) {  
 hostPatchProp(el, "style", oldProps.style, newProps.style, namespace);  
 }  
 if (patchFlag & 8) {  
 const propsToUpdate = n2.dynamicProps;  
 for (let i = 0; i < propsToUpdate.length; i++) {  
 const key = propsToUpdate[i];  
 const prev = oldProps[key];  
 const next = newProps[key];  
 if (next !== prev || key === "value") {  
 hostPatchProp(el, key, prev, next, namespace, parentComponent);  
 }  
 }  
 }  
 }  
 if (patchFlag & 1) {  
 if (n1.children !== n2.children) {  
 hostSetElementText(el, n2.children);  
 }  
 }  
 } else if (!optimized && dynamicChildren == null) {  
 patchProps(el, oldProps, newProps, parentComponent, namespace);  
 }  
 if ((vnodeHook = newProps.onVnodeUpdated) || dirs) {  
 queuePostRenderEffect(() => {  
 vnodeHook && invokeVNodeHook(vnodeHook, parentComponent, n2, n1);  
 dirs && invokeDirectiveHook(n2, n1, parentComponent, "updated");  
 }, parentSuspense);  
 }  
 };  
 const patchBlockChildren = (oldChildren, newChildren, fallbackContainer, parentComponent, parentSuspense, namespace, slotScopeIds) => {  
 for (let i = 0; i < newChildren.length; i++) {  
 const oldVNode = oldChildren[i];  
 const newVNode = newChildren[i];  
 const container = (  
 oldVNode.el && // - In the case of a Fragment, we need to provide the actual parent  
 (oldVNode.type === Fragment || // - In the case of different nodes, there is going to be a replacement  
 !isSameVNodeType(oldVNode, newVNode) || // - In the case of a component, it could contain anything.  
 oldVNode.shapeFlag & (6 | 64)) ? hostParentNode(oldVNode.el) : (  
 fallbackContainer  
 )  
 );  
 patch(  
 oldVNode,  
 newVNode,  
 container,  
 null,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 true  
 );  
 }  
 };  
 const patchProps = (el, oldProps, newProps, parentComponent, namespace) => {  
 if (oldProps !== newProps) {  
 if (oldProps !== EMPTY\_OBJ) {  
 for (const key in oldProps) {  
 if (!isReservedProp(key) && !(key in newProps)) {  
 hostPatchProp(  
 el,  
 key,  
 oldProps[key],  
 null,  
 namespace,  
 parentComponent  
 );  
 }  
 }  
 }  
 for (const key in newProps) {  
 if (isReservedProp(key)) continue;  
 const next = newProps[key];  
 const prev = oldProps[key];  
 if (next !== prev && key !== "value") {  
 hostPatchProp(el, key, prev, next, namespace, parentComponent);  
 }  
 }  
 if ("value" in newProps) {  
 hostPatchProp(el, "value", oldProps.value, newProps.value, namespace);  
 }  
 }  
 };  
 const processFragment = (n1, n2, container, anchor, parentComponent, parentSuspense, namespace, slotScopeIds, optimized) => {  
 const fragmentStartAnchor = n2.el = n1 ? n1.el : hostCreateText("");  
 const fragmentEndAnchor = n2.anchor = n1 ? n1.anchor : hostCreateText("");  
 let { patchFlag, dynamicChildren, slotScopeIds: fragmentSlotScopeIds } = n2;  
 if (  
 isHmrUpdating || patchFlag & 2048  
 ) {  
 patchFlag = 0;  
 optimized = false;  
 dynamicChildren = null;  
 }  
 if (fragmentSlotScopeIds) {  
 slotScopeIds = slotScopeIds ? slotScopeIds.concat(fragmentSlotScopeIds) : fragmentSlotScopeIds;  
 }  
 if (n1 == null) {  
 hostInsert(fragmentStartAnchor, container, anchor);  
 hostInsert(fragmentEndAnchor, container, anchor);  
 mountChildren(  
 n2.children || [],  
 container,  
 fragmentEndAnchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 } else {  
  
 n1.dynamicChildren) {  
 patchBlockChildren(  
 n1.dynamicChildren,  
 dynamicChildren,  
 container,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds  
 );  
 if (true) {  
 traverseStaticChildren(n1, n2);  
 } else if (  
 n2.key != null || parentComponent && n2 === parentComponent.subTree  
 ) {  
 traverseStaticChildren(  
 n1,  
 n2,  
 true  
 );  
 }  
 } else {  
 patchChildren(  
 n1,  
 n2,  
 container,  
 fragmentEndAnchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 }  
 }  
 };  
 const processComponent = (n1, n2, container, anchor, parentComponent, parentSuspense, namespace, slotScopeIds, optimized) => {  
 n2.slotScopeIds = slotScopeIds;  
 if (n1 == null) {  
 if (n2.shapeFlag & 512) {  
 parentComponent.ctx.activate(  
 n2,  
 container,  
 anchor,  
 namespace,  
 optimized  
 );  
 } else {  
 mountComponent(  
 n2,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 optimized  
 );  
 }  
 } else {  
 updateComponent(n1, n2, optimized);  
 }  
 };  
 const mountComponent = (initialVNode, container, anchor, parentComponent, parentSuspense, namespace, optimized) => {  
 const instance = initialVNode.component = createComponentInstance(  
 initialVNode,  
 parentComponent,  
 parentSuspense  
 );  
 if (instance.type.\_\_hmrId) {  
 registerHMR(instance);  
 }  
 if (true) {  
 pushWarningContext(initialVNode);  
 startMeasure(instance, `mount`);  
 }  
 if (isKeepAlive(initialVNode)) {  
 instance.ctx.renderer = internals;  
 }  
 {  
 if (true) {  
 startMeasure(instance, `init`);  
 }  
 setupComponent(instance, false, optimized);  
 if (true) {  
 endMeasure(instance, `init`);  
 }  
 }  
 if (instance.asyncDep) {  
 parentSuspense && parentSuspense.registerDep(instance, setupRenderEffect, optimized);  
 if (!initialVNode.el) {  
 const placeholder = instance.subTree = createVNode(Comment);  
 processCommentNode(null, placeholder, container, anchor);  
 }  
 } else {  
 setupRenderEffect(  
 instance,  
 initialVNode,  
 container,  
 anchor,  
 parentSuspense,  
 namespace,  
 optimized  
 );  
 }  
 if (true) {  
 popWarningContext();  
 endMeasure(instance, `mount`);  
 }  
 };  
 const updateComponent = (n1, n2, optimized) => {  
 const instance = n2.component = n1.component;  
 if (shouldUpdateComponent(n1, n2, optimized)) {  
 if (instance.asyncDep && !instance.asyncResolved) {  
 if (true) {  
 pushWarningContext(n2);  
 }  
 updateComponentPreRender(instance, n2, optimized);  
 if (true) {  
 popWarningContext();  
 }  
 return;  
 } else {  
 instance.next = n2;  
 invalidateJob(instance.update);  
 instance.effect.dirty = true;  
 instance.update();  
 }  
 } else {  
 n2.el = n1.el;  
 instance.vnode = n2;  
 }  
 };  
 const setupRenderEffect = (instance, initialVNode, container, anchor, parentSuspense, namespace, optimized) => {  
 const componentUpdateFn = () => {  
 if (!instance.isMounted) {  
 let vnodeHook;  
 const { el, props } = initialVNode;  
 const { bm, m, parent } = instance;  
 const isAsyncWrapperVNode = isAsyncWrapper(initialVNode);  
 toggleRecurse(instance, false);  
 if (bm) {  
 invokeArrayFns(bm);  
 }  
 if (!isAsyncWrapperVNode && (vnodeHook = props && props.onVnodeBeforeMount)) {  
 invokeVNodeHook(vnodeHook, parent, initialVNode);  
 }  
 toggleRecurse(instance, true);  
 if (el && hydrateNode) {  
 const hydrateSubTree = () => {  
 if (true) {  
 startMeasure(instance, `render`);  
 }  
 instance.subTree = renderComponentRoot(instance);  
 if (true) {  
 endMeasure(instance, `render`);  
 }  
 if (true) {  
 startMeasure(instance, `hydrate`);  
 }  
 hydrateNode(  
 el,  
 instance.subTree,  
 instance,  
 parentSuspense,  
 null  
 );  
 if (true) {  
 endMeasure(instance, `hydrate`);  
 }  
 };  
 if (isAsyncWrapperVNode) {  
 initialVNode.type.\_\_asyncLoader().then(  
 () => !instance.isUnmounted && hydrateSubTree()  
 );  
 } else {  
 hydrateSubTree();  
 }  
 } else {  
 if (true) {  
 startMeasure(instance, `render`);  
 }  
 const subTree = instance.subTree = renderComponentRoot(instance);  
 if (true) {  
 endMeasure(instance, `render`);  
 }  
 if (true) {  
 startMeasure(instance, `patch`);  
 }  
 patch(  
 null,  
 subTree,  
 container,  
 anchor,  
 instance,  
 parentSuspense,  
 namespace  
 );  
 if (true) {  
 endMeasure(instance, `patch`);  
 }  
 initialVNode.el = subTree.el;  
 }  
 if (m) {  
 queuePostRenderEffect(m, parentSuspense);  
 }  
 if (!isAsyncWrapperVNode && (vnodeHook = props && props.onVnodeMounted)) {  
 const scopedInitialVNode = initialVNode;  
 queuePostRenderEffect(  
 () => invokeVNodeHook(vnodeHook, parent, scopedInitialVNode),  
 parentSuspense  
 );  
 }  
  
 instance.a && queuePostRenderEffect(instance.a, parentSuspense);  
 }  
 instance.isMounted = true;  
 if (true) {  
 devtoolsComponentAdded(instance);  
 }  
 initialVNode = container = anchor = null;  
 } else {  
 let { next, bu, u, parent, vnode } = instance;  
 {  
 const nonHydratedAsyncRoot = locateNonHydratedAsyncRoot(instance);  
 if (nonHydratedAsyncRoot) {  
 if (next) {  
 next.el = vnode.el;  
 updateComponentPreRender(instance, next, optimized);  
 }  
 nonHydratedAsyncRoot.asyncDep.then(() => {  
 if (!instance.isUnmounted) {  
 componentUpdateFn();  
 }  
 });  
 return;  
 }  
 }  
 let originNext = next;  
 let vnodeHook;  
 if (true) {  
 pushWarningContext(next || instance.vnode);  
 }  
 toggleRecurse(instance, false);  
 if (next) {  
 next.el = vnode.el;  
 updateComponentPreRender(instance, next, optimized);  
 } else {  
 next = vnode;  
 }  
 if (bu) {  
 invokeArrayFns(bu);  
 }  
 if (vnodeHook = next.props && next.props.onVnodeBeforeUpdate) {  
 invokeVNodeHook(vnodeHook, parent, next, vnode);  
 }  
 toggleRecurse(instance, true);  
 if (true) {  
 startMeasure(instance, `render`);  
 }  
 const nextTree = renderComponentRoot(instance);  
 if (true) {  
 endMeasure(instance, `render`);  
 }  
 const prevTree = instance.subTree;  
 instance.subTree = nextTree;  
 if (true) {  
 startMeasure(instance, `patch`);  
 }  
 patch(  
 prevTree,  
 nextTree,  
 hostParentNode(prevTree.el),  
 getNextHostNode(prevTree),  
 instance,  
 parentSuspense,  
 namespace  
 );  
 if (true) {  
 endMeasure(instance, `patch`);  
 }  
 next.el = nextTree.el;  
 if (originNext === null) {  
 updateHOCHostEl(instance, nextTree.el);  
 }  
 if (u) {  
 queuePostRenderEffect(u, parentSuspense);  
 }  
 if (vnodeHook = next.props && next.props.onVnodeUpdated) {  
 queuePostRenderEffect(  
 () => invokeVNodeHook(vnodeHook, parent, next, vnode),  
 parentSuspense  
 );  
 }  
 if (true) {  
 devtoolsComponentUpdated(instance);  
 }  
 if (true) {  
 popWarningContext();  
 }  
 }  
 };  
 const effect2 = instance.effect = new ReactiveEffect(  
 componentUpdateFn,  
 NOOP,  
 () => queueJob(update),  
 instance.scope  
 );  
 const update = instance.update = () => {  
 if (effect2.dirty) {  
 effect2.run();  
 }  
 };  
 update.i = instance;  
 update.id = instance.uid;  
 toggleRecurse(instance, true);  
 if (true) {  
 effect2.onTrack = instance.rtc ? (e) => invokeArrayFns(instance.rtc, e) : void 0;  
 effect2.onTrigger = instance.rtg ? (e) => invokeArrayFns(instance.rtg, e) : void 0;  
 }  
 update();  
 };  
 const updateComponentPreRender = (instance, nextVNode, optimized) => {  
 nextVNode.component = instance;  
 const prevProps = instance.vnode.props;  
 instance.vnode = nextVNode;  
 instance.next = null;  
 updateProps(instance, nextVNode.props, prevProps, optimized);  
 updateSlots(instance, nextVNode.children, optimized);  
 pauseTracking();  
 flushPreFlushCbs(instance);  
 resetTracking();  
 };  
 const patchChildren = (n1, n2, container, anchor, parentComponent, parentSuspense, namespace, slotScopeIds, optimized = false) => {  
 const c1 = n1 && n1.children;  
 const prevShapeFlag = n1 ? n1.shapeFlag : 0;  
 const c2 = n2.children;  
 const { patchFlag, shapeFlag } = n2;  
 if (patchFlag > 0) {  
 if (patchFlag & 128) {  
 patchKeyedChildren(  
 c1,  
 c2,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 return;  
 } else if (patchFlag & 256) {  
 patchUnkeyedChildren(  
 c1,  
 c2,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 return;  
 }  
 }  
 if (shapeFlag & 8) {  
 if (prevShapeFlag & 16) {  
 unmountChildren(c1, parentComponent, parentSuspense);  
 }  
 if (c2 !== c1) {  
 hostSetElementText(container, c2);  
 }  
 } else {  
 if (prevShapeFlag & 16) {  
 if (shapeFlag & 16) {  
 patchKeyedChildren(  
 c1,  
 c2,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 } else {  
 unmountChildren(c1, parentComponent, parentSuspense, true);  
 }  
 } else {  
 if (prevShapeFlag & 8) {  
 hostSetElementText(container, "");  
 }  
 if (shapeFlag & 16) {  
 mountChildren(  
 c2,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 }  
 }  
 }  
 };  
 const patchUnkeyedChildren = (c1, c2, container, anchor, parentComponent, parentSuspense, namespace, slotScopeIds, optimized) => {  
 c1 = c1 || EMPTY\_ARR;  
 c2 = c2 || EMPTY\_ARR;  
 const oldLength = c1.length;  
 const newLength = c2.length;  
 const commonLength = Math.min(oldLength, newLength);  
 let i;  
 for (i = 0; i < commonLength; i++) {  
 const nextChild = c2[i] = optimized ? cloneIfMounted(c2[i]) : normalizeVNode(c2[i]);  
 patch(  
 c1[i],  
 nextChild,  
 container,  
 null,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 }  
 if (oldLength > newLength) {  
 unmountChildren(  
 c1,  
 parentComponent,  
 parentSuspense,  
 true,  
 false,  
 commonLength  
 );  
 } else {  
 mountChildren(  
 c2,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized,  
 commonLength  
 );  
 }  
 };  
 const patchKeyedChildren = (c1, c2, container, parentAnchor, parentComponent, parentSuspense, namespace, slotScopeIds, optimized) => {  
 let i = 0;  
 const l2 = c2.length;  
 let e1 = c1.length - 1;  
 let e2 = l2 - 1;  
 while (i <= e1 && i <= e2) {  
 const n1 = c1[i];  
 const n2 = c2[i] = optimized ? cloneIfMounted(c2[i]) : normalizeVNode(c2[i]);  
 if (isSameVNodeType(n1, n2)) {  
 patch(  
 n1,  
 n2,  
 container,  
 null,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 } else {  
 break;  
 }  
 i++;  
 }  
 while (i <= e1 && i <= e2) {  
 const n1 = c1[e1];  
 const n2 = c2[e2] = optimized ? cloneIfMounted(c2[e2]) : normalizeVNode(c2[e2]);  
 if (isSameVNodeType(n1, n2)) {  
 patch(  
 n1,  
 n2,  
 container,  
 null,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 } else {  
 break;  
 }  
 e1--;  
 e2--;  
 }  
 if (i > e1) {  
 if (i <= e2) {  
 const nextPos = e2 + 1;  
 const anchor = nextPos < l2 ? c2[nextPos].el : parentAnchor;  
 while (i <= e2) {  
 patch(  
 null,  
 c2[i] = optimized ? cloneIfMounted(c2[i]) : normalizeVNode(c2[i]),  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 i++;  
 }  
 }  
 } else if (i > e2) {  
 while (i <= e1) {  
 unmount(c1[i], parentComponent, parentSuspense, true);  
 i++;  
 }  
 } else {  
 const s1 = i;  
 const s2 = i;  
 const keyToNewIndexMap = /\* @\_\_PURE\_\_ \*/ new Map();  
 for (i = s2; i <= e2; i++) {  
 const nextChild = c2[i] = optimized ? cloneIfMounted(c2[i]) : normalizeVNode(c2[i]);  
 if (nextChild.key != null) {  
 if (keyToNewIndexMap.has(nextChild.key)) {  
 warn$1(  
 `Duplicate keys found during update:`,  
 JSON.stringify(nextChild.key),  
 `Make sure keys are unique.`  
 );  
 }  
 keyToNewIndexMap.set(nextChild.key, i);  
 }  
 }  
 let j;  
 let patched = 0;  
 const toBePatched = e2 - s2 + 1;  
 let moved = false;  
 let maxNewIndexSoFar = 0;  
 const newIndexToOldIndexMap = new Array(toBePatched);  
 for (i = 0; i < toBePatched; i++) newIndexToOldIndexMap[i] = 0;  
 for (i = s1; i <= e1; i++) {  
 const prevChild = c1[i];  
 if (patched >= toBePatched) {  
 unmount(prevChild, parentComponent, parentSuspense, true);  
 continue;  
 }  
 let newIndex;  
 if (prevChild.key != null) {  
 newIndex = keyToNewIndexMap.get(prevChild.key);  
 } else {  
 for (j = s2; j <= e2; j++) {  
 if (newIndexToOldIndexMap[j - s2] === 0 && isSameVNodeType(prevChild, c2[j])) {  
 newIndex = j;  
 break;  
 }  
 }  
 }  
 if (newIndex === void 0) {  
 unmount(prevChild, parentComponent, parentSuspense, true);  
 } else {  
 newIndexToOldIndexMap[newIndex - s2] = i + 1;  
 if (newIndex >= maxNewIndexSoFar) {  
 maxNewIndexSoFar = newIndex;  
 } else {  
 moved = true;  
 }  
 patch(  
 prevChild,  
 c2[newIndex],  
 container,  
 null,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 patched++;  
 }  
 }  
 const increasingNewIndexSequence = moved ? getSequence(newIndexToOldIndexMap) : EMPTY\_ARR;  
 j = increasingNewIndexSequence.length - 1;  
 for (i = toBePatched - 1; i >= 0; i--) {  
 const nextIndex = s2 + i;  
 const nextChild = c2[nextIndex];  
 const anchor = nextIndex + 1 < l2 ? c2[nextIndex + 1].el : parentAnchor;  
 if (newIndexToOldIndexMap[i] === 0) {  
 patch(  
 null,  
 nextChild,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 } else if (moved) {  
 if (j < 0 || i !== increasingNewIndexSequence[j]) {  
 move(nextChild, container, anchor, 2);  
 } else {  
 j--;  
 }  
 }  
 }  
 }  
 };  
 const move = (vnode, container, anchor, moveType, parentSuspense = null) => {  
 const { el, type, transition, children, shapeFlag } = vnode;  
 if (shapeFlag & 6) {  
 move(vnode.component.subTree, container, anchor, moveType);  
 return;  
 }  
 if (shapeFlag & 128) {  
 vnode.suspense.move(container, anchor, moveType);  
 return;  
 }  
 if (shapeFlag & 64) {  
 type.move(vnode, container, anchor, internals);  
 return;  
 }  
 if (type === Fragment) {  
 hostInsert(el, container, anchor);  
 for (let i = 0; i < children.length; i++) {  
 move(children[i], container, anchor, moveType);  
 }  
 hostInsert(vnode.anchor, container, anchor);  
 return;  
 }  
 if (type === Static) {  
 moveStaticNode(vnode, container, anchor);  
 return;  
 }  
 const needTransition2 = moveType !== 2 && shapeFlag & 1 && transition;  
 if (needTransition2) {  
 if (moveType === 0) {  
 transition.beforeEnter(el);  
 hostInsert(el, container, anchor);  
 queuePostRenderEffect(() => transition.enter(el), parentSuspense);  
 } else {  
 const { leave, delayLeave, afterLeave } = transition;  
 const remove22 = () => hostInsert(el, container, anchor);  
 const performLeave = () => {  
 leave(el, () => {  
 remove22();  
 afterLeave && afterLeave();  
 });  
 };  
 if (delayLeave) {  
 delayLeave(el, remove22, performLeave);  
 } else {  
 performLeave();  
 }  
 }  
 } else {  
 hostInsert(el, container, anchor);  
 }  
 };  
 const unmount = (vnode, parentComponent, parentSuspense, doRemove = false, optimized = false) => {  
 const {  
 type,  
 props,  
 ref: ref2,  
 children,  
 dynamicChildren,  
 shapeFlag,  
 patchFlag,  
 dirs,  
 cacheIndex  
 } = vnode;  
 if (patchFlag === -2) {  
 optimized = false;  
 }  
 if (ref2 != null) {  
 setRef(ref2, null, parentSuspense, vnode, true);  
 }  
 if (cacheIndex != null) {  
 parentComponent.renderCache[cacheIndex] = void 0;  
 }  
 if (shapeFlag & 256) {  
 parentComponent.ctx.deactivate(vnode);  
 return;  
 }  
 const shouldInvokeDirs = shapeFlag & 1 && dirs;  
 const shouldInvokeVnodeHook = !isAsyncWrapper(vnode);  
 let vnodeHook;  
 if (shouldInvokeVnodeHook && (vnodeHook = props && props.onVnodeBeforeUnmount)) {  
 invokeVNodeHook(vnodeHook, parentComponent, vnode);  
 }  
 if (shapeFlag & 6) {  
 unmountComponent(vnode.component, parentSuspense, doRemove);  
 } else {  
 if (shapeFlag & 128) {  
 vnode.suspense.unmount(parentSuspense, doRemove);  
 return;  
 }  
 if (shouldInvokeDirs) {  
 invokeDirectiveHook(vnode, null, parentComponent, "beforeUnmount");  
 }  
 if (shapeFlag & 64) {  
 vnode.type.remove(  
 vnode,  
 parentComponent,  
 parentSuspense,  
 internals,  
 doRemove  
 );  
 } else if (dynamicChildren && // #5154  
 !dynamicChildren.hasOnce && // #1153: fast path should not be taken for non-stable (v-for) fragments  
 (type !== Fragment || patchFlag > 0 && patchFlag & 64)) {  
 unmountChildren(  
 dynamicChildren,  
 parentComponent,  
 parentSuspense,  
 false,  
 true  
 );  
  
 unmountChildren(children, parentComponent, parentSuspense);  
 }  
 if (doRemove) {  
 remove2(vnode);  
 }  
 }  
 if (shouldInvokeVnodeHook && (vnodeHook = props && props.onVnodeUnmounted) || shouldInvokeDirs) {  
 queuePostRenderEffect(() => {  
 vnodeHook && invokeVNodeHook(vnodeHook, parentComponent, vnode);  
 shouldInvokeDirs && invokeDirectiveHook(vnode, null, parentComponent, "unmounted");  
 }, parentSuspense);  
 }  
 };  
 const remove2 = (vnode) => {  
 const { type, el, anchor, transition } = vnode;  
 if (type === Fragment) {  
  
 vnode.children.forEach((child) => {  
 if (child.type === Comment) {  
 hostRemove(child.el);  
 } else {  
 remove2(child);  
 }  
 });  
 } else {  
 removeFragment(el, anchor);  
 }  
 return;  
 }  
 if (type === Static) {  
 removeStaticNode(vnode);  
 return;  
 }  
 const performRemove = () => {  
 hostRemove(el);  
 if (transition && !transition.persisted && transition.afterLeave) {  
 transition.afterLeave();  
 }  
 };  
 if (vnode.shapeFlag & 1 && transition && !transition.persisted) {  
 const { leave, delayLeave } = transition;  
 const performLeave = () => leave(el, performRemove);  
 if (delayLeave) {  
 delayLeave(vnode.el, performRemove, performLeave);  
 } else {  
 performLeave();  
 }  
 } else {  
 performRemove();  
 }  
 };  
 const removeFragment = (cur, end) => {  
 let next;  
 while (cur !== end) {  
 next = hostNextSibling(cur);  
 hostRemove(cur);  
 cur = next;  
 }  
 hostRemove(end);  
 };  
 const unmountComponent = (instance, parentSuspense, doRemove) => {  
 if (instance.type.\_\_hmrId) {  
 unregisterHMR(instance);  
 }  
 const { bum, scope, update, subTree, um, m, a } = instance;  
 invalidateMount(m);  
 invalidateMount(a);  
 if (bum) {  
 invokeArrayFns(bum);  
 }  
 scope.stop();  
 if (update) {  
 update.active = false;  
 unmount(subTree, instance, parentSuspense, doRemove);  
 }  
 if (um) {  
 queuePostRenderEffect(um, parentSuspense);  
 }  
 queuePostRenderEffect(() => {  
 instance.isUnmounted = true;  
 }, parentSuspense);  
  
 parentSuspense.deps--;  
 if (parentSuspense.deps === 0) {  
 parentSuspense.resolve();  
 }  
 }  
 if (true) {  
 devtoolsComponentRemoved(instance);  
 }  
 };  
 const unmountChildren = (children, parentComponent, parentSuspense, doRemove = false, optimized = false, start = 0) => {  
 for (let i = start; i < children.length; i++) {  
 unmount(children[i], parentComponent, parentSuspense, doRemove, optimized);  
 }  
 };  
 const getNextHostNode = (vnode) => {  
 if (vnode.shapeFlag & 6) {  
 return getNextHostNode(vnode.component.subTree);  
 }  
 if (vnode.shapeFlag & 128) {  
 return vnode.suspense.next();  
 }  
 const el = hostNextSibling(vnode.anchor || vnode.el);  
 const teleportEnd = el && el[TeleportEndKey];  
 return teleportEnd ? hostNextSibling(teleportEnd) : el;  
 };  
 let isFlushing2 = false;  
 const render2 = (vnode, container, namespace) => {  
 if (vnode == null) {  
 if (container.\_vnode) {  
 unmount(container.\_vnode, null, null, true);  
 }  
 } else {  
 patch(  
 container.\_vnode || null,  
 vnode,  
 container,  
 null,  
 null,  
 null,  
 namespace  
 );  
 }  
 if (!isFlushing2) {  
 isFlushing2 = true;  
 flushPreFlushCbs();  
 flushPostFlushCbs();  
 isFlushing2 = false;  
 }  
 container.\_vnode = vnode;  
 };  
 const internals = {  
 p: patch,  
 um: unmount,  
 m: move,  
 r: remove2,  
 mt: mountComponent,  
 mc: mountChildren,  
 pc: patchChildren,  
 pbc: patchBlockChildren,  
 n: getNextHostNode,  
 o: options  
 };  
 let hydrate2;  
 let hydrateNode;  
 if (createHydrationFns) {  
 [hydrate2, hydrateNode] = createHydrationFns(  
 internals  
 );  
 }  
 return {  
 render: render2,  
 hydrate: hydrate2,  
 createApp: createAppAPI(render2, hydrate2)  
 };  
}  
function resolveChildrenNamespace({ type, props }, currentNamespace) {  
  
}  
function toggleRecurse({ effect: effect2, update }, allowed) {  
 effect2.allowRecurse = update.allowRecurse = allowed;  
}  
function needTransition(parentSuspense, transition) {  
  
}  
function traverseStaticChildren(n1, n2, shallow = false) {  
 const ch1 = n1.children;  
 const ch2 = n2.children;  
 if (isArray(ch1) && isArray(ch2)) {  
 for (let i = 0; i < ch1.length; i++) {  
 const c1 = ch1[i];  
 let c2 = ch2[i];  
 if (c2.shapeFlag & 1 && !c2.dynamicChildren) {  
 if (c2.patchFlag <= 0 || c2.patchFlag === 32) {  
 c2 = ch2[i] = cloneIfMounted(ch2[i]);  
 c2.el = c1.el;  
 }  
 if (!shallow && c2.patchFlag !== -2)  
 traverseStaticChildren(c1, c2);  
 }  
 if (c2.type === Text) {  
 c2.el = c1.el;  
 }  
 if (c2.type === Comment && !c2.el) {  
 c2.el = c1.el;  
 }  
 }  
 }  
}  
function getSequence(arr) {  
 const p2 = arr.slice();  
 const result = [0];  
 let i, j, u, v, c;  
 const len = arr.length;  
 for (i = 0; i < len; i++) {  
 const arrI = arr[i];  
 if (arrI !== 0) {  
 j = result[result.length - 1];  
 if (arr[j] < arrI) {  
 p2[i] = j;  
 result.push(i);  
 continue;  
 }  
 u = 0;  
 v = result.length - 1;  
 while (u < v) {  
 c = u + v >> 1;  
 if (arr[result[c]] < arrI) {  
 u = c + 1;  
 } else {  
 v = c;  
 }  
 }  
 if (arrI < arr[result[u]]) {  
 if (u > 0) {  
 p2[i] = result[u - 1];  
 }  
 result[u] = i;  
 }  
 }  
 }  
 u = result.length;  
 v = result[u - 1];  
 while (u-- > 0) {  
 result[u] = v;  
 v = p2[v];  
 }  
 return result;  
}  
function locateNonHydratedAsyncRoot(instance) {  
 const subComponent = instance.subTree.component;  
 if (subComponent) {  
 if (subComponent.asyncDep && !subComponent.asyncResolved) {  
 return subComponent;  
 } else {  
 return locateNonHydratedAsyncRoot(subComponent);  
 }  
 }  
}  
function invalidateMount(hooks) {  
 if (hooks) {  
 for (let i = 0; i < hooks.length; i++) hooks[i].active = false;  
 }  
}  
var ssrContextKey = Symbol.for("v-scx");  
var useSSRContext = () => {  
 {  
 const ctx = inject(ssrContextKey);  
 if (!ctx) {  
 warn$1(  
 `Server rendering context not provided. Make sure to only call useSSRContext() conditionally in the server build.`  
 );  
 }  
 return ctx;  
 }  
};  
function watchEffect(effect2, options) {  
 return doWatch(effect2, null, options);  
}  
function watchPostEffect(effect2, options) {  
 return doWatch(  
 effect2,  
 null,  
 true ? extend({}, options, { flush: "post" }) : { flush: "post" }  
 );  
}  
function watchSyncEffect(effect2, options) {  
 return doWatch(  
 effect2,  
 null,  
 true ? extend({}, options, { flush: "sync" }) : { flush: "sync" }  
 );  
}  
var INITIAL\_WATCHER\_VALUE = {};  
function watch(source, cb, options) {  
 if (!isFunction(cb)) {  
 warn$1(  
 `\`watch(fn, options?)\` signature has been moved to a separate API. Use \`watchEffect(fn, options?)\` instead. \`watch\` now only supports \`watch(source, cb, options?) signature.`  
 );  
 }  
 return doWatch(source, cb, options);  
}  
function doWatch(source, cb, {  
 immediate,  
 deep,  
 flush,  
 once,  
 onTrack,  
 onTrigger  
} = EMPTY\_OBJ) {  
 if (cb && once) {  
 const \_cb = cb;  
 cb = (...args) => {  
 \_cb(...args);  
 unwatch();  
 };  
 }  
 if (deep !== void 0 && typeof deep === "number") {  
 warn$1(  
 `watch() "deep" option with number value will be used as watch depth in future versions. Please use a boolean instead to avoid potential breakage.`  
 );  
 }  
 if (!cb) {  
 if (immediate !== void 0) {  
 warn$1(  
 `watch() "immediate" option is only respected when using the watch(source, callback, options?) signature.`  
 );  
 }  
 if (deep !== void 0) {  
 warn$1(  
 `watch() "deep" option is only respected when using the watch(source, callback, options?) signature.`  
 );  
 }  
 if (once !== void 0) {  
 warn$1(  
 `watch() "once" option is only respected when using the watch(source, callback, options?) signature.`  
 );  
 }  
 }  
 const warnInvalidSource = (s) => {  
 warn$1(  
 `Invalid watch source: `,  
 s,  
 `A watch source can only be a getter/effect function, a ref, a reactive object, or an array of these types.`  
 );  
 };  
 const instance = currentInstance;  
 const reactiveGetter = (source2) => deep === true ? source2 : (  
 traverse(source2, deep === false ? 1 : void 0)  
 );  
 let getter;  
 let forceTrigger = false;  
 let isMultiSource = false;  
 if (isRef2(source)) {  
 getter = () => source.value;  
 forceTrigger = isShallow(source);  
 } else if (isReactive(source)) {  
 getter = () => reactiveGetter(source);  
 forceTrigger = true;  
 } else if (isArray(source)) {  
 isMultiSource = true;  
 forceTrigger = source.some((s) => isReactive(s) || isShallow(s));  
 getter = () => source.map((s) => {  
 if (isRef2(s)) {  
 return s.value;  
 } else if (isReactive(s)) {  
 return reactiveGetter(s);  
 } else if (isFunction(s)) {  
 return callWithErrorHandling(s, instance, 2);  
 } else {  
 warnInvalidSource(s);  
 }  
 });  
 } else if (isFunction(source)) {  
 if (cb) {  
 getter = () => callWithErrorHandling(source, instance, 2);  
 } else {  
 getter = () => {  
 if (cleanup) {  
 cleanup();  
 }  
 return callWithAsyncErrorHandling(  
 source,  
 instance,  
 3,  
 [onCleanup]  
 );  
 };  
 }  
 } else {  
 getter = NOOP;  
 warnInvalidSource(source);  
 }  
 if (cb && deep) {  
 const baseGetter = getter;  
 getter = () => traverse(baseGetter());  
 }  
 let cleanup;  
 let onCleanup = (fn) => {  
 cleanup = effect2.onStop = () => {  
 callWithErrorHandling(fn, instance, 4);  
 cleanup = effect2.onStop = void 0;  
 };  
 };  
 let ssrCleanup;  
 if (isInSSRComponentSetup) {  
 onCleanup = NOOP;  
 if (!cb) {  
 getter();  
 } else if (immediate) {  
 callWithAsyncErrorHandling(cb, instance, 3, [  
 getter(),  
 isMultiSource ? [] : void 0,  
 onCleanup  
 ]);  
 }  
 if (flush === "sync") {  
 const ctx = useSSRContext();  
 ssrCleanup = ctx.\_\_watcherHandles || (ctx.\_\_watcherHandles = []);  
 } else {  
 return NOOP;  
 }  
 }  
 let oldValue = isMultiSource ? new Array(source.length).fill(INITIAL\_WATCHER\_VALUE) : INITIAL\_WATCHER\_VALUE;  
 const job = () => {  
 if (!effect2.active || !effect2.dirty) {  
 return;  
 }  
 if (cb) {  
 const newValue = effect2.run();  
 if (deep || forceTrigger || (isMultiSource ? newValue.some((v, i) => hasChanged(v, oldValue[i])) : hasChanged(newValue, oldValue)) || false) {  
 if (cleanup) {  
 cleanup();  
 }  
 callWithAsyncErrorHandling(cb, instance, 3, [  
 newValue,  
 oldValue === INITIAL\_WATCHER\_VALUE ? void 0 : isMultiSource && oldValue[0] === INITIAL\_WATCHER\_VALUE ? [] : oldValue,  
 onCleanup  
 ]);  
 oldValue = newValue;  
 }  
 } else {  
 effect2.run();  
 }  
 };  
 job.allowRecurse = !!cb;  
 let scheduler;  
 if (flush === "sync") {  
 scheduler = job;  
 } else if (flush === "post") {  
 scheduler = () => queuePostRenderEffect(job, instance && instance.suspense);  
 } else {  
 job.pre = true;  
 if (instance) job.id = instance.uid;  
 scheduler = () => queueJob(job);  
 }  
 const effect2 = new ReactiveEffect(getter, NOOP, scheduler);  
 const scope = getCurrentScope();  
 const unwatch = () => {  
 effect2.stop();  
 if (scope) {  
 remove(scope.effects, effect2);  
 }  
 };  
 if (true) {  
 effect2.onTrack = onTrack;  
 effect2.onTrigger = onTrigger;  
 }  
 if (cb) {  
 if (immediate) {  
 job();  
 } else {  
 oldValue = effect2.run();  
 }  
 } else if (flush === "post") {  
 queuePostRenderEffect(  
 effect2.run.bind(effect2),  
 instance && instance.suspense  
 );  
 } else {  
 effect2.run();  
 }  
 if (ssrCleanup) ssrCleanup.push(unwatch);  
 return unwatch;  
}  
function instanceWatch(source, value, options) {  
 const publicThis = this.proxy;  
 const getter = isString(source) ? source.includes(".") ? createPathGetter(publicThis, source) : () => publicThis[source] : source.bind(publicThis, publicThis);  
 let cb;  
 if (isFunction(value)) {  
 cb = value;  
 } else {  
 cb = value.handler;  
 options = value;  
 }  
 const reset = setCurrentInstance(this);  
 const res = doWatch(getter, cb.bind(publicThis), options);  
 reset();  
 return res;  
}  
function createPathGetter(ctx, path) {  
 const segments = path.split(".");  
 return () => {  
 let cur = ctx;  
 for (let i = 0; i < segments.length && cur; i++) {  
 cur = cur[segments[i]];  
 }  
 return cur;  
 };  
}  
function traverse(value, depth = Infinity, seen) {  
 if (depth <= 0 || !isObject(value) || value["\_\_v\_skip"]) {  
 return value;  
 }  
 seen = seen || /\* @\_\_PURE\_\_ \*/ new Set();  
 if (seen.has(value)) {  
 return value;  
 }  
 seen.add(value);  
 depth--;  
 if (isRef2(value)) {  
 traverse(value.value, depth, seen);  
 } else if (isArray(value)) {  
 for (let i = 0; i < value.length; i++) {  
 traverse(value[i], depth, seen);  
 }  
 } else if (isSet(value) || isMap(value)) {  
 value.forEach((v) => {  
 traverse(v, depth, seen);  
 });  
 } else if (isPlainObject(value)) {  
 for (const key in value) {  
 traverse(value[key], depth, seen);  
 }  
 for (const key of Object.getOwnPropertySymbols(value)) {  
 if (Object.prototype.propertyIsEnumerable.call(value, key)) {  
 traverse(value[key], depth, seen);  
 }  
 }  
 }  
 return value;  
}  
function useModel(props, name, options = EMPTY\_OBJ) {  
 const i = getCurrentInstance();  
 if (!i) {  
 warn$1(`useModel() called without active instance.`);  
 return ref();  
 }  
 if (!i.propsOptions[0][name]) {  
 warn$1(`useModel() called with prop "${name}" which is not declared.`);  
 return ref();  
 }  
 const camelizedName = camelize(name);  
 const hyphenatedName = hyphenate(name);  
 const modifiers = getModelModifiers(props, name);  
 const res = customRef((track2, trigger2) => {  
 let localValue;  
 let prevSetValue;  
 let prevEmittedValue;  
 watchSyncEffect(() => {  
 const propValue = props[name];  
 if (hasChanged(localValue, propValue)) {  
 localValue = propValue;  
 trigger2();  
 }  
 });  
 return {  
 get() {  
 track2();  
 return options.get ? options.get(localValue) : localValue;  
 },  
 set(value) {  
 if (!hasChanged(value, localValue)) {  
 return;  
 }  
 const rawProps = i.vnode.props;  
 if (!(rawProps && // check if parent has passed v-model  
 (name in rawProps || camelizedName in rawProps || hyphenatedName in rawProps) && (`onUpdate:${name}` in rawProps || `onUpdate:${camelizedName}` in rawProps || `onUpdate:${hyphenatedName}` in rawProps))) {  
 localValue = value;  
 trigger2();  
 }  
 const emittedValue = options.set ? options.set(value) : value;  
 i.emit(`update:${name}`, emittedValue);  
 if (value !== emittedValue && value !== prevSetValue && emittedValue === prevEmittedValue) {  
 trigger2();  
 }  
 prevSetValue = value;  
 prevEmittedValue = emittedValue;  
 }  
 };  
 });  
 res[Symbol.iterator] = () => {  
 let i2 = 0;  
 return {  
 next() {  
 if (i2 < 2) {  
 return { value: i2++ ? modifiers || EMPTY\_OBJ : res, done: false };  
 } else {  
 return { done: true };  
 }  
 }  
 };  
 };  
 return res;  
}  
var getModelModifiers = (props, modelName) => {  
 return modelName === "modelValue" || modelName === "model-value" ? props.modelModifiers : props[`${modelName}Modifiers`] || props[`${camelize(modelName)}Modifiers`] || props[`${hyphenate(modelName)}Modifiers`];  
};  
function emit(instance, event, ...rawArgs) {  
 if (instance.isUnmounted) return;  
 const props = instance.vnode.props || EMPTY\_OBJ;  
 if (true) {  
 const {  
 emitsOptions,  
 propsOptions: [propsOptions]  
 } = instance;  
 if (emitsOptions) {  
 if (!(event in emitsOptions) && true) {  
 if (!propsOptions || !(toHandlerKey(event) in propsOptions)) {  
 warn$1(  
 `Component emitted event "${event}" but it is neither declared in the emits option nor as an "${toHandlerKey(event)}" prop.`  
 );  
 }  
 } else {  
 const validator = emitsOptions[event];  
 if (isFunction(validator)) {  
 const isValid = validator(...rawArgs);  
 if (!isValid) {  
 warn$1(  
 `Invalid event arguments: event validation failed for event "${event}".`  
 );  
 }  
 }  
 }  
 }  
 }  
 let args = rawArgs;  
 const isModelListener2 = event.startsWith("update:");  
 const modifiers = isModelListener2 && getModelModifiers(props, event.slice(7));  
 if (modifiers) {  
 if (modifiers.trim) {  
 args = rawArgs.map((a) => isString(a) ? a.trim() : a);  
 }  
 if (modifiers.number) {  
 args = rawArgs.map(looseToNumber);  
 }  
 }  
 if (true) {  
 devtoolsComponentEmit(instance, event, args);  
 }  
 if (true) {  
 const lowerCaseEvent = event.toLowerCase();  
 if (lowerCaseEvent !== event && props[toHandlerKey(lowerCaseEvent)]) {  
 warn$1(  
 `Event "${lowerCaseEvent}" is emitted in component ${formatComponentName(  
 instance,  
 instance.type  
 )} but the handler is registered for "${event}". Note that HTML attributes are case-insensitive and you cannot use v-on to listen to camelCase events when using in-DOM templates. You should probably use "${hyphenate(  
 event  
 )}" instead of "${event}".`  
 );  
 }  
 }  
 let handlerName;  
 let handler = props[handlerName = toHandlerKey(event)] || // also try camelCase event handler (#2249)  
 props[handlerName = toHandlerKey(camelize(event))];  
 if (!handler && isModelListener2) {  
 handler = props[handlerName = toHandlerKey(hyphenate(event))];  
 }  
 if (handler) {  
 callWithAsyncErrorHandling(  
 handler,  
 instance,  
 6,  
 args  
 );  
 }  
 const onceHandler = props[handlerName + `Once`];  
 if (onceHandler) {  
 if (!instance.emitted) {  
 instance.emitted = {};  
 } else if (instance.emitted[handlerName]) {  
 return;  
 }  
 instance.emitted[handlerName] = true;  
 callWithAsyncErrorHandling(  
 onceHandler,  
 instance,  
 6,  
 args  
 );  
 }  
}  
function normalizeEmitsOptions(comp, appContext, asMixin = false) {  
 const cache = appContext.emitsCache;  
 const cached = cache.get(comp);  
 if (cached !== void 0) {  
 return cached;  
 }  
 const raw = comp.emits;  
 let normalized = {};  
 let hasExtends = false;  
 if (\_\_VUE\_OPTIONS\_API\_\_ && !isFunction(comp)) {  
 const extendEmits = (raw2) => {  
 const normalizedFromExtend = normalizeEmitsOptions(raw2, appContext, true);  
 if (normalizedFromExtend) {  
 hasExtends = true;  
 extend(normalized, normalizedFromExtend);  
 }  
 };  
 if (!asMixin && appContext.mixins.length) {  
 appContext.mixins.forEach(extendEmits);  
 }  
 if (comp.extends) {  
 extendEmits(comp.extends);  
 }  
 if (comp.mixins) {  
 comp.mixins.forEach(extendEmits);  
 }  
 }  
 if (!raw && !hasExtends) {  
 if (isObject(comp)) {  
 cache.set(comp, null);  
 }  
 return null;  
 }  
 if (isArray(raw)) {  
 raw.forEach((key) => normalized[key] = null);  
 } else {  
 extend(normalized, raw);  
 }  
 if (isObject(comp)) {  
 cache.set(comp, normalized);  
 }  
 return normalized;  
}  
function isEmitListener(options, key) {  
 if (!options || !isOn(key)) {  
 return false;  
 }  
 key = key.slice(2).replace(/Once$/, "");  
 return hasOwn(options, key[0].toLowerCase() + key.slice(1)) || hasOwn(options, hyphenate(key)) || hasOwn(options, key);  
}  
var accessedAttrs = false;  
function markAttrsAccessed() {  
 accessedAttrs = true;  
}  
function renderComponentRoot(instance) {  
 const {  
 type: Component,  
 vnode,  
 proxy,  
 withProxy,  
 propsOptions: [propsOptions],  
 slots,  
 attrs,  
 emit: emit2,  
 render: render2,  
 renderCache,  
 props,  
 data,  
 setupState,  
 ctx,  
 inheritAttrs  
 } = instance;  
 const prev = setCurrentRenderingInstance(instance);  
 let result;  
 let fallthroughAttrs;  
 if (true) {  
 accessedAttrs = false;  
 }  
 try {  
 if (vnode.shapeFlag & 4) {  
 const proxyToUse = withProxy || proxy;  
 const thisProxy = setupState.\_\_isScriptSetup ? new Proxy(proxyToUse, {  
 get(target, key, receiver) {  
 warn$1(  
 `Property '${String(  
 key  
 )}' was accessed via 'this'. Avoid using 'this' in templates.`  
 );  
 return Reflect.get(target, key, receiver);  
 }  
 }) : proxyToUse;  
 result = normalizeVNode(  
 render2.call(  
 thisProxy,  
 proxyToUse,  
 renderCache,  
 true ? shallowReadonly(props) : props,  
 setupState,  
 data,  
 ctx  
 )  
 );  
 fallthroughAttrs = attrs;  
 } else {  
 const render22 = Component;  
 if (attrs === props) {  
 markAttrsAccessed();  
 }  
 result = normalizeVNode(  
 render22.length > 1 ? render22(  
 true ? shallowReadonly(props) : props,  
 true ? {  
 get attrs() {  
 markAttrsAccessed();  
 return shallowReadonly(attrs);  
 },  
 slots,  
 emit: emit2  
 } : { attrs, slots, emit: emit2 }  
 ) : render22(  
 true ? shallowReadonly(props) : props,  
 null  
 )  
 );  
 fallthroughAttrs = Component.props ? attrs : getFunctionalFallthrough(attrs);  
 }  
 } catch (err) {  
 blockStack.length = 0;  
 handleError(err, instance, 1);  
 result = createVNode(Comment);  
 }  
 let root = result;  
 let setRoot = void 0;  
 if (result.patchFlag > 0 && result.patchFlag & 2048) {  
 [root, setRoot] = getChildRoot(result);  
 }  
 if (fallthroughAttrs && inheritAttrs !== false) {  
 const keys = Object.keys(fallthroughAttrs);  
 const { shapeFlag } = root;  
 if (keys.length) {  
 if (shapeFlag & (1 | 6)) {  
 if (propsOptions && keys.some(isModelListener)) {  
 fallthroughAttrs = filterModelListeners(  
 fallthroughAttrs,  
 propsOptions  
 );  
 }  
 root = cloneVNode(root, fallthroughAttrs, false, true);  
 } else if (!accessedAttrs && root.type !== Comment) {  
 const allAttrs = Object.keys(attrs);  
 const eventAttrs = [];  
 const extraAttrs = [];  
 for (let i = 0, l = allAttrs.length; i < l; i++) {  
 const key = allAttrs[i];  
 if (isOn(key)) {  
 if (!isModelListener(key)) {  
 eventAttrs.push(key[2].toLowerCase() + key.slice(3));  
 }  
 } else {  
 extraAttrs.push(key);  
 }  
 }  
 if (extraAttrs.length) {  
 warn$1(  
 `Extraneous non-props attributes (${extraAttrs.join(", ")}) were passed to component but could not be automatically inherited because component renders fragment or text root nodes.`  
 );  
 }  
 if (eventAttrs.length) {  
 warn$1(  
 `Extraneous non-emits event listeners (${eventAttrs.join(", ")}) were passed to component but could not be automatically inherited because component renders fragment or text root nodes. If the listener is intended to be a component custom event listener only, declare it using the "emits" option.`  
 );  
 }  
 }  
 }  
 }  
 if (vnode.dirs) {  
 if (!isElementRoot(root)) {  
 warn$1(  
 `Runtime directive used on component with non-element root node. The directives will not function as intended.`  
 );  
 }  
 root = cloneVNode(root, null, false, true);  
 root.dirs = root.dirs ? root.dirs.concat(vnode.dirs) : vnode.dirs;  
 }  
 if (vnode.transition) {  
 if (!isElementRoot(root)) {  
 warn$1(  
 `Component inside <Transition> renders non-element root node that cannot be animated.`  
 );  
 }  
 root.transition = vnode.transition;  
 }  
 if (setRoot) {  
 setRoot(root);  
 } else {  
 result = root;  
 }  
 setCurrentRenderingInstance(prev);  
 return result;  
}  
var getChildRoot = (vnode) => {  
 const rawChildren = vnode.children;  
 const dynamicChildren = vnode.dynamicChildren;  
 const childRoot = filterSingleRoot(rawChildren, false);  
 if (!childRoot) {  
 return [vnode, void 0];  
 } else if (childRoot.patchFlag > 0 && childRoot.patchFlag & 2048) {  
 return getChildRoot(childRoot);  
 }  
 const index = rawChildren.indexOf(childRoot);  
 const dynamicIndex = dynamicChildren ? dynamicChildren.indexOf(childRoot) : -1;  
 const setRoot = (updatedRoot) => {  
 rawChildren[index] = updatedRoot;  
 if (dynamicChildren) {  
 if (dynamicIndex > -1) {  
 dynamicChildren[dynamicIndex] = updatedRoot;  
 } else if (updatedRoot.patchFlag > 0) {  
 vnode.dynamicChildren = [...dynamicChildren, updatedRoot];  
 }  
 }  
 };  
 return [normalizeVNode(childRoot), setRoot];  
};  
function filterSingleRoot(children, recurse = true) {  
 let singleRoot;  
 for (let i = 0; i < children.length; i++) {  
 const child = children[i];  
 if (isVNode(child)) {  
 if (child.type !== Comment || child.children === "v-if") {  
 if (singleRoot) {  
 return;  
 } else {  
 singleRoot = child;  
 if (recurse && singleRoot.patchFlag > 0 && singleRoot.patchFlag & 2048) {  
 return filterSingleRoot(singleRoot.children);  
 }  
 }  
 }  
 } else {  
 return;  
 }  
 }  
 return singleRoot;  
}  
var getFunctionalFallthrough = (attrs) => {  
 let res;  
 for (const key in attrs) {  
 if (key === "class" || key === "style" || isOn(key)) {  
 (res || (res = {}))[key] = attrs[key];  
 }  
 }  
 return res;  
};  
var filterModelListeners = (attrs, props) => {  
 const res = {};  
 for (const key in attrs) {  
 if (!isModelListener(key) || !(key.slice(9) in props)) {  
 res[key] = attrs[key];  
 }  
 }  
 return res;  
};  
var isElementRoot = (vnode) => {  
 return vnode.shapeFlag & (6 | 1) || vnode.type === Comment;  
};  
function shouldUpdateComponent(prevVNode, nextVNode, optimized) {  
 const { props: prevProps, children: prevChildren, component } = prevVNode;  
 const { props: nextProps, children: nextChildren, patchFlag } = nextVNode;  
 const emits = component.emitsOptions;  
 if ((prevChildren || nextChildren) && isHmrUpdating) {  
 return true;  
 }  
 if (nextVNode.dirs || nextVNode.transition) {  
 return true;  
 }  
 if (optimized && patchFlag >= 0) {  
 if (patchFlag & 1024) {  
 return true;  
 }  
 if (patchFlag & 16) {  
 if (!prevProps) {  
 return !!nextProps;  
 }  
 return hasPropsChanged(prevProps, nextProps, emits);  
 } else if (patchFlag & 8) {  
 const dynamicProps = nextVNode.dynamicProps;  
 for (let i = 0; i < dynamicProps.length; i++) {  
 const key = dynamicProps[i];  
 if (nextProps[key] !== prevProps[key] && !isEmitListener(emits, key)) {  
 return true;  
 }  
 }  
 }  
 } else {  
 if (prevChildren || nextChildren) {  
 if (!nextChildren || !nextChildren.$stable) {  
 return true;  
 }  
 }  
 if (prevProps === nextProps) {  
 return false;  
 }  
 if (!prevProps) {  
 return !!nextProps;  
 }  
 if (!nextProps) {  
 return true;  
 }  
 return hasPropsChanged(prevProps, nextProps, emits);  
 }  
 return false;  
}  
function hasPropsChanged(prevProps, nextProps, emitsOptions) {  
 const nextKeys = Object.keys(nextProps);  
 if (nextKeys.length !== Object.keys(prevProps).length) {  
 return true;  
 }  
 for (let i = 0; i < nextKeys.length; i++) {  
 const key = nextKeys[i];  
 if (nextProps[key] !== prevProps[key] && !isEmitListener(emitsOptions, key)) {  
 return true;  
 }  
 }  
 return false;  
}  
function updateHOCHostEl({ vnode, parent }, el) {  
 while (parent) {  
 const root = parent.subTree;  
 if (root.suspense && root.suspense.activeBranch === vnode) {  
 root.el = vnode.el;  
 }  
 if (root === vnode) {  
 (vnode = parent.vnode).el = el;  
 parent = parent.parent;  
 } else {  
 break;  
 }  
 }  
}  
var isSuspense = (type) => type.\_\_isSuspense;  
var suspenseId = 0;  
var SuspenseImpl = {  
 name: "Suspense",  
 \_\_isSuspense: true,  
 process(n1, n2, container, anchor, parentComponent, parentSuspense, namespace, slotScopeIds, optimized, rendererInternals) {  
 if (n1 == null) {  
 mountSuspense(  
 n2,  
 container,  
 anchor,  
 parentComponent,  
 parentSuspense,  
 namespace,  
 slotScopeIds,  
 optimized,  
 rendererInternals  
 );  
 } else {  
 if (parentSuspense && parentSuspense.deps > 0 && !n1.suspense.isInFallback) {  
 n2.suspense = n1.suspense;  
 n2.suspense.vnode = n2;  
 n2.el = n1.el;  
 return;  
 }  
 patchSuspense(  
 n1,  
 n2,  
 container,  
 anchor,  
 parentComponent,  
 namespace,  
 slotScopeIds,  
 optimized,  
 rendererInternals  
 );  
 }  
 },  
 hydrate: hydrateSuspense,  
 normalize: normalizeSuspenseChildren  
};  
var Suspense = SuspenseImpl;  
function triggerEvent(vnode, name) {  
 const eventListener = vnode.props && vnode.props[name];  
 if (isFunction(eventListener)) {  
 eventListener();  
 }  
}  
function mountSuspense(vnode, container, anchor, parentComponent, parentSuspense, namespace, slotScopeIds, optimized, rendererInternals) {  
 const {  
 p: patch,  
 o: { createElement }  
 } = rendererInternals;  
 const hiddenContainer = createElement("div");  
 const suspense = vnode.suspense = createSuspenseBoundary(  
 vnode,  
 parentSuspense,  
 parentComponent,  
 container,  
 hiddenContainer,  
 anchor,  
 namespace,  
 slotScopeIds,  
 optimized,  
 rendererInternals  
 );  
 patch(  
 null,  
 suspense.pendingBranch = vnode.ssContent,  
 hiddenContainer,  
 null,  
 parentComponent,  
 suspense,  
 namespace,  
 slotScopeIds  
 );  
 if (suspense.deps > 0) {  
 triggerEvent(vnode, "onPending");  
 triggerEvent(vnode, "onFallback");  
 patch(  
 null,  
 vnode.ssFallback,  
 container,  
 anchor,  
 parentComponent,  
 null,  
 namespace,  
 slotScopeIds  
 );  
 setActiveBranch(suspense, vnode.ssFallback);  
 } else {  
 suspense.resolve(false, true);  
 }  
}  
function patchSuspense(n1, n2, container, anchor, parentComponent, namespace, slotScopeIds, optimized, { p: patch, um: unmount, o: { createElement } }) {  
 const suspense = n2.suspense = n1.suspense;  
 suspense.vnode = n2;  
 n2.el = n1.el;  
 const newBranch = n2.ssContent;  
 const newFallback = n2.ssFallback;  
 const { activeBranch, pendingBranch, isInFallback, isHydrating } = suspense;  
 if (pendingBranch) {  
 suspense.pendingBranch = newBranch;  
 if (isSameVNodeType(newBranch, pendingBranch)) {  
 patch(  
 pendingBranch,  
 newBranch,  
 suspense.hiddenContainer,  
 null,  
 parentComponent,  
 suspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 if (suspense.deps <= 0) {  
 suspense.resolve();  
 } else if (isInFallback) {  
 if (!isHydrating) {  
 patch(  
 activeBranch,  
 newFallback,  
 container,  
 anchor,  
 parentComponent,  
 null,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 setActiveBranch(suspense, newFallback);  
 }  
 }  
 } else {  
 suspense.pendingId = suspenseId++;  
 if (isHydrating) {  
 suspense.isHydrating = false;  
 suspense.activeBranch = pendingBranch;  
 } else {  
 unmount(pendingBranch, parentComponent, suspense);  
 }  
 suspense.deps = 0;  
 suspense.effects.length = 0;  
 suspense.hiddenContainer = createElement("div");  
 if (isInFallback) {  
 patch(  
 null,  
 newBranch,  
 suspense.hiddenContainer,  
 null,  
 parentComponent,  
 suspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 if (suspense.deps <= 0) {  
 suspense.resolve();  
 } else {  
 patch(  
 activeBranch,  
 newFallback,  
 container,  
 anchor,  
 parentComponent,  
 null,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 setActiveBranch(suspense, newFallback);  
 }  
 } else if (activeBranch && isSameVNodeType(newBranch, activeBranch)) {  
 patch(  
 activeBranch,  
 newBranch,  
 container,  
 anchor,  
 parentComponent,  
 suspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 suspense.resolve(true);  
 } else {  
 patch(  
 null,  
 newBranch,  
 suspense.hiddenContainer,  
 null,  
 parentComponent,  
 suspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 if (suspense.deps <= 0) {  
 suspense.resolve();  
 }  
 }  
 }  
 } else {  
 if (activeBranch && isSameVNodeType(newBranch, activeBranch)) {  
 patch(  
 activeBranch,  
 newBranch,  
 container,  
 anchor,  
 parentComponent,  
 suspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 setActiveBranch(suspense, newBranch);  
 } else {  
 triggerEvent(n2, "onPending");  
 suspense.pendingBranch = newBranch;  
 if (newBranch.shapeFlag & 512) {  
 suspense.pendingId = newBranch.component.suspenseId;  
 } else {  
 suspense.pendingId = suspenseId++;  
 }  
 patch(  
 null,  
 newBranch,  
 suspense.hiddenContainer,  
 null,  
 parentComponent,  
 suspense,  
 namespace,  
 slotScopeIds,  
 optimized  
 );  
 if (suspense.deps <= 0) {  
 suspense.resolve();  
 } else {  
 const { timeout, pendingId } = suspense;  
 if (timeout > 0) {  
 setTimeout(() => {  
 if (suspense.pendingId === pendingId) {  
 suspense.fallback(newFallback);  
 }  
 }, timeout);  
 } else if (timeout === 0) {  
 suspense.fallback(newFallback);  
 }  
 }  
 }  
 }  
}  
var hasWarned = false;  
function createSuspenseBoundary(vnode, parentSuspense, parentComponent, container, hiddenContainer, anchor, namespace, slotScopeIds, optimized, rendererInternals, isHydrating = false) {  
 if (!hasWarned) {  
 hasWarned = true;  
 console[console.info ? "info" : "log"](  
 `<Suspense> is an experimental feature and its API will likely change.`  
 );  
 }  
 const {  
 p: patch,  
 m: move,  
 um: unmount,  
 n: next,  
 o: { parentNode, remove: remove2 }  
 } = rendererInternals;  
 let parentSuspenseId;  
 const isSuspensible = isVNodeSuspensible(vnode);  
 if (isSuspensible) {  
 if (parentSuspense && parentSuspense.pendingBranch) {  
 parentSuspenseId = parentSuspense.pendingId;  
 parentSuspense.deps++;  
 }  
 }  
 const timeout = vnode.props ? toNumber(vnode.props.timeout) : void 0;  
 if (true) {  
 assertNumber(timeout, `Suspense timeout`);  
 }  
 const initialAnchor = anchor;  
 const suspense = {  
 vnode,  
 parent: parentSuspense,  
 parentComponent,  
 namespace,  
 container,  
 hiddenContainer,  
 deps: 0,  
 pendingId: suspenseId++,  
 timeout: typeof timeout === "number" ? timeout : -1,  
 activeBranch: null,  
 pendingBranch: null,  
 isInFallback: !isHydrating,  
 isHydrating,  
 isUnmounted: false,  
 effects: [],  
 resolve(resume = false, sync = false) {  
 if (true) {  
 if (!resume && !suspense.pendingBranch) {  
 throw new Error(  
 `suspense.resolve() is called without a pending branch.`  
 );  
 }  
 if (suspense.isUnmounted) {  
 throw new Error(  
 `suspense.resolve() is called on an already unmounted suspense boundary.`  
 );  
 }  
 }  
 const {  
 vnode: vnode2,  
 activeBranch,  
 pendingBranch,  
 pendingId,  
 effects,  
 parentComponent: parentComponent2,  
 container: container2  
 } = suspense;  
 let delayEnter = false;  
 if (suspense.isHydrating) {  
 suspense.isHydrating = false;  
 } else if (!resume) {  
 delayEnter = activeBranch && pendingBranch.transition && pendingBranch.transition.mode === "out-in";  
 if (delayEnter) {  
 activeBranch.transition.afterLeave = () => {  
 if (pendingId === suspense.pendingId) {  
 move(  
 pendingBranch,  
 container2,  
 anchor === initialAnchor ? next(activeBranch) : anchor,  
 0  
 );  
 queuePostFlushCb(effects);  
 }  
 };  
 }  
 if (activeBranch) {  
 if (parentNode(activeBranch.el) !== suspense.hiddenContainer) {  
 anchor = next(activeBranch);  
 }  
 unmount(activeBranch, parentComponent2, suspense, true);  
 }  
 if (!delayEnter) {  
 move(pendingBranch, container2, anchor, 0);  
 }  
 }  
 setActiveBranch(suspense, pendingBranch);  
 suspense.pendingBranch = null;  
 suspense.isInFallback = false;  
 let parent = suspense.parent;  
 let hasUnresolvedAncestor = false;  
 while (parent) {  
 if (parent.pendingBranch) {  
 parent.effects.push(...effects);  
 hasUnresolvedAncestor = true;  
 break;  
 }  
 parent = parent.parent;  
 }  
 if (!hasUnresolvedAncestor && !delayEnter) {  
 queuePostFlushCb(effects);  
 }  
 suspense.effects = [];  
 if (isSuspensible) {  
 if (parentSuspense && parentSuspense.pendingBranch && parentSuspenseId === parentSuspense.pendingId) {  
 parentSuspense.deps--;  
 if (parentSuspense.deps === 0 && !sync) {  
 parentSuspense.resolve();  
 }  
 }  
 }  
 triggerEvent(vnode2, "onResolve");  
 },  
 fallback(fallbackVNode) {  
 if (!suspense.pendingBranch) {  
 return;  
 }  
 const { vnode: vnode2, activeBranch, parentComponent: parentComponent2, container: container2, namespace: namespace2 } = suspense;  
 triggerEvent(vnode2, "onFallback");  
 const anchor2 = next(activeBranch);  
 const mountFallback = () => {  
 if (!suspense.isInFallback) {  
 return;  
 }  
 patch(  
 null,  
 fallbackVNode,  
 container2,  
 anchor2,  
 parentComponent2,  
 null,  
 namespace2,  
 slotScopeIds,  
 optimized  
 );  
 setActiveBranch(suspense, fallbackVNode);  
 };  
 const delayEnter = fallbackVNode.transition && fallbackVNode.transition.mode === "out-in";  
 if (delayEnter) {  
 activeBranch.transition.afterLeave = mountFallback;  
 }  
 suspense.isInFallback = true;  
 unmount(  
 activeBranch,  
 parentComponent2,  
 null,  
 true  
 );  
 if (!delayEnter) {  
 mountFallback();  
 }  
 },  
 move(container2, anchor2, type) {  
 suspense.activeBranch && move(suspense.activeBranch, container2, anchor2, type);  
 suspense.container = container2;  
 },  
 next() {  
 return suspense.activeBranch && next(suspense.activeBranch);  
 },  
 registerDep(instance, setupRenderEffect, optimized2) {  
 const isInPendingSuspense = !!suspense.pendingBranch;  
 if (isInPendingSuspense) {  
 suspense.deps++;  
 }  
 const hydratedEl = instance.vnode.el;  
 instance.asyncDep.catch((err) => {  
 handleError(err, instance, 0);  
 }).then((asyncSetupResult) => {  
 if (instance.isUnmounted || suspense.isUnmounted || suspense.pendingId !== instance.suspenseId) {  
 return;  
 }  
 instance.asyncResolved = true;  
 const { vnode: vnode2 } = instance;  
 if (true) {  
 pushWarningContext(vnode2);  
 }  
 handleSetupResult(instance, asyncSetupResult, false);  
 if (hydratedEl) {  
 vnode2.el = hydratedEl;  
 }  
 const placeholder = !hydratedEl && instance.subTree.el;  
 setupRenderEffect(  
 instance,  
 vnode2,  
 parentNode(hydratedEl || instance.subTree.el),  
 hydratedEl ? null : next(instance.subTree),  
 suspense,  
 namespace,  
 optimized2  
 );  
 if (placeholder) {  
 remove2(placeholder);  
 }  
 updateHOCHostEl(instance, vnode2.el);  
 if (true) {  
 popWarningContext();  
 }  
 if (isInPendingSuspense && --suspense.deps === 0) {  
 suspense.resolve();  
 }  
 });  
 },  
 unmount(parentSuspense2, doRemove) {  
 suspense.isUnmounted = true;  
 if (suspense.activeBranch) {  
 unmount(  
 suspense.activeBranch,  
 parentComponent,  
 parentSuspense2,  
 doRemove  
 );  
 }  
 if (suspense.pendingBranch) {  
 unmount(  
 suspense.pendingBranch,  
 parentComponent,  
 parentSuspense2,  
 doRemove  
 );  
 }  
 }  
 };  
 return suspense;  
}  
function hydrateSuspense(node, vnode, parentComponent, parentSuspense, namespace, slotScopeIds, optimized, rendererInternals, hydrateNode) {  
 const suspense = vnode.suspense = createSuspenseBoundary(  
 vnode,  
 parentSuspense,  
 parentComponent,  
 node.parentNode,  
 document.createElement("div"),  
 null,  
 namespace,  
 slotScopeIds,  
 optimized,  
 rendererInternals,  
 true  
 );  
 const result = hydrateNode(  
 node,  
 suspense.pendingBranch = vnode.ssContent,  
 parentComponent,  
 suspense,  
 slotScopeIds,  
 optimized  
 );  
 if (suspense.deps === 0) {  
 suspense.resolve(false, true);  
 }  
 return result;  
}  
function normalizeSuspenseChildren(vnode) {  
 const { shapeFlag, children } = vnode;  
 const isSlotChildren = shapeFlag & 32;  
 vnode.ssContent = normalizeSuspenseSlot(  
 isSlotChildren ? children.default : children  
 );  
 vnode.ssFallback = isSlotChildren ? normalizeSuspenseSlot(children.fallback) : createVNode(Comment);  
}  
function normalizeSuspenseSlot(s) {  
 let block;  
 if (isFunction(s)) {  
 const trackBlock = isBlockTreeEnabled && s.\_c;  
 if (trackBlock) {  
 s.\_d = false;  
 openBlock();  
 }  
 s = s();  
 if (trackBlock) {  
 s.\_d = true;  
 block = currentBlock;  
 closeBlock();  
 }  
 }  
 if (isArray(s)) {  
 const singleChild = filterSingleRoot(s);  
 if (!singleChild && s.filter((child) => child !== NULL\_DYNAMIC\_COMPONENT).length > 0) {  
 warn$1(`<Suspense> slots expect a single root node.`);  
 }  
 s = singleChild;  
 }  
 s = normalizeVNode(s);  
 if (block && !s.dynamicChildren) {  
 s.dynamicChildren = block.filter((c) => c !== s);  
 }  
 return s;  
}  
function queueEffectWithSuspense(fn, suspense) {  
 if (suspense && suspense.pendingBranch) {  
 if (isArray(fn)) {  
 suspense.effects.push(...fn);  
 } else {  
 suspense.effects.push(fn);  
 }  
 } else {  
 queuePostFlushCb(fn);  
 }  
}  
function setActiveBranch(suspense, branch) {  
 suspense.activeBranch = branch;  
 const { vnode, parentComponent } = suspense;  
 let el = branch.el;  
 while (!el && branch.component) {  
 branch = branch.component.subTree;  
 el = branch.el;  
 }  
 vnode.el = el;  
 if (parentComponent && parentComponent.subTree === vnode) {  
 parentComponent.vnode.el = el;  
 updateHOCHostEl(parentComponent, el);  
 }  
}  
function isVNodeSuspensible(vnode) {  
 const suspensible = vnode.props && vnode.props.suspensible;  
 return suspensible != null && suspensible !== false;  
}  
var Fragment = Symbol.for("v-fgt");  
var Text = Symbol.for("v-txt");  
var Comment = Symbol.for("v-cmt");  
var Static = Symbol.for("v-stc");  
var blockStack = [];  
var currentBlock = null;  
function openBlock(disableTracking = false) {  
 blockStack.push(currentBlock = disableTracking ? null : []);  
}  
function closeBlock() {  
 blockStack.pop();  
 currentBlock = blockStack[blockStack.length - 1] || null;  
}  
var isBlockTreeEnabled = 1;  
function setBlockTracking(value) {  
 isBlockTreeEnabled += value;  
 if (value < 0 && currentBlock) {  
 currentBlock.hasOnce = true;  
 }  
}  
function setupBlock(vnode) {  
 vnode.dynamicChildren = isBlockTreeEnabled > 0 ? currentBlock || EMPTY\_ARR : null;  
 closeBlock();  
 if (isBlockTreeEnabled > 0 && currentBlock) {  
 currentBlock.push(vnode);  
 }  
 return vnode;  
}  
function createElementBlock(type, props, children, patchFlag, dynamicProps, shapeFlag) {  
 return setupBlock(  
 createBaseVNode(  
 type,  
 props,  
 children,  
 patchFlag,  
 dynamicProps,  
 shapeFlag,  
 true  
 )  
 );  
}  
function createBlock(type, props, children, patchFlag, dynamicProps) {  
 return setupBlock(  
 createVNode(  
 type,  
 props,  
 children,  
 patchFlag,  
 dynamicProps,  
 true  
 )  
 );  
}  
function isVNode(value) {  
 return value ? value.\_\_v\_isVNode === true : false;  
}  
function isSameVNodeType(n1, n2) {  
 if (n2.shapeFlag & 6 && n1.component) {  
 const dirtyInstances = hmrDirtyComponents.get(n2.type);  
 if (dirtyInstances && dirtyInstances.has(n1.component)) {  
 n1.shapeFlag &= ~256;  
 n2.shapeFlag &= ~512;  
 return false;  
 }  
 }  
 return n1.type === n2.type && n1.key === n2.key;  
}  
var vnodeArgsTransformer;  
function transformVNodeArgs(transformer) {  
 vnodeArgsTransformer = transformer;  
}  
var createVNodeWithArgsTransform = (...args) => {  
 return \_createVNode(  
 ...vnodeArgsTransformer ? vnodeArgsTransformer(args, currentRenderingInstance) : args  
 );  
};  
var normalizeKey = ({ key }) => key != null ? key : null;  
var normalizeRef = ({  
 ref: ref2,  
 ref\_key,  
 ref\_for  
}) => {  
 if (typeof ref2 === "number") {  
 ref2 = "" + ref2;  
 }  
 return ref2 != null ? isString(ref2) || isRef2(ref2) || isFunction(ref2) ? { i: currentRenderingInstance, r: ref2, k: ref\_key, f: !!ref\_for } : ref2 : null;  
};  
function createBaseVNode(type, props = null, children = null, patchFlag = 0, dynamicProps = null, shapeFlag = type === Fragment ? 0 : 1, isBlockNode = false, needFullChildrenNormalization = false) {  
 const vnode = {  
 \_\_v\_isVNode: true,  
 \_\_v\_skip: true,  
 type,  
 props,  
 key: props && normalizeKey(props),  
 ref: props && normalizeRef(props),  
 scopeId: currentScopeId,  
 slotScopeIds: null,  
 children,  
 component: null,  
 suspense: null,  
 ssContent: null,  
 ssFallback: null,  
 dirs: null,  
 transition: null,  
 el: null,  
 anchor: null,  
 target: null,  
 targetStart: null,  
 targetAnchor: null,  
 staticCount: 0,  
 shapeFlag,  
 patchFlag,  
 dynamicProps,  
 dynamicChildren: null,  
 appContext: null,  
 ctx: currentRenderingInstance  
 };  
 if (needFullChildrenNormalization) {  
 normalizeChildren(vnode, children);  
 if (shapeFlag & 128) {  
 type.normalize(vnode);  
 }  
 } else if (children) {  
 vnode.shapeFlag |= isString(children) ? 8 : 16;  
 }  
 if (vnode.key !== vnode.key) {  
 warn$1(`VNode created with invalid key (NaN). VNode type:`, vnode.type);  
 }  
 if (isBlockTreeEnabled > 0 && // avoid a block node from tracking itself  
 !isBlockNode && // has current parent block  
 currentBlock && // presence of a patch flag indicates this node needs patching on updates.  
 (vnode.patchFlag > 0 || shapeFlag & 6) && // the EVENTS flag is only for hydration and if it is the only flag, the  
 vnode.patchFlag !== 32) {  
 currentBlock.push(vnode);  
 }  
 return vnode;  
}  
var createVNode = true ? createVNodeWithArgsTransform : \_createVNode;  
function \_createVNode(type, props = null, children = null, patchFlag = 0, dynamicProps = null, isBlockNode = false) {  
 if (!type || type === NULL\_DYNAMIC\_COMPONENT) {  
 if (!type) {  
 warn$1(`Invalid vnode type when creating vnode: ${type}.`);  
 }  
 type = Comment;  
 }  
 if (isVNode(type)) {  
 const cloned = cloneVNode(  
 type,  
 props,  
 true  
 );  
 if (children) {  
 normalizeChildren(cloned, children);  
 }  
 if (isBlockTreeEnabled > 0 && !isBlockNode && currentBlock) {  
 if (cloned.shapeFlag & 6) {  
 currentBlock[currentBlock.indexOf(type)] = cloned;  
 } else {  
 currentBlock.push(cloned);  
 }  
 }  
 cloned.patchFlag = -2;  
 return cloned;  
 }  
 if (isClassComponent(type)) {  
 type = type.\_\_vccOpts;  
 }  
 if (props) {  
 props = guardReactiveProps(props);  
 let { class: klass, style } = props;  
 if (klass && !isString(klass)) {  
 props.class = normalizeClass(klass);  
 }  
 if (isObject(style)) {  
 if (isProxy(style) && !isArray(style)) {  
 style = extend({}, style);  
 }  
 props.style = normalizeStyle(style);  
 }  
 }  
 const shapeFlag = isString(type) ? 1 : isSuspense(type) ? 128 : isTeleport(type) ? 64 : isObject(type) ? 4 : isFunction(type) ? 2 : 0;  
 if (shapeFlag & 4 && isProxy(type)) {  
 type = toRaw(type);  
 warn$1(  
 `Vue received a Component that was made a reactive object. This can lead to unnecessary performance overhead and should be avoided by marking the component with \`markRaw\` or using \`shallowRef\` instead of \`ref\`.`,  
 `  
Component that was made reactive: `,  
 type  
 );  
 }  
 return createBaseVNode(  
 type,  
 props,  
 children,  
 patchFlag,  
 dynamicProps,  
 shapeFlag,  
 isBlockNode,  
 true  
 );  
}  
function guardReactiveProps(props) {  
 if (!props) return null;  
 return isProxy(props) || isInternalObject(props) ? extend({}, props) : props;  
}  
function cloneVNode(vnode, extraProps, mergeRef = false, cloneTransition = false) {  
 const { props, ref: ref2, patchFlag, children, transition } = vnode;  
 const mergedProps = extraProps ? mergeProps(props || {}, extraProps) : props;  
 const cloned = {  
 \_\_v\_isVNode: true,  
 \_\_v\_skip: true,  
 type: vnode.type,  
 props: mergedProps,  
 key: mergedProps && normalizeKey(mergedProps),  
 ref: extraProps && extraProps.ref ? (  
 mergeRef && ref2 ? isArray(ref2) ? ref2.concat(normalizeRef(extraProps)) : [ref2, normalizeRef(extraProps)] : normalizeRef(extraProps)  
 ) : ref2,  
 scopeId: vnode.scopeId,  
 slotScopeIds: vnode.slotScopeIds,  
 children: patchFlag === -1 && isArray(children) ? children.map(deepCloneVNode) : children,  
 target: vnode.target,  
 targetStart: vnode.targetStart,  
 targetAnchor: vnode.targetAnchor,  
 staticCount: vnode.staticCount,  
 shapeFlag: vnode.shapeFlag,  
 patchFlag: extraProps && vnode.type !== Fragment ? patchFlag === -1 ? 16 : patchFlag | 16 : patchFlag,  
 dynamicProps: vnode.dynamicProps,  
 dynamicChildren: vnode.dynamicChildren,  
 appContext: vnode.appContext,  
 dirs: vnode.dirs,  
 transition,  
 component: vnode.component,  
 suspense: vnode.suspense,  
 ssContent: vnode.ssContent && cloneVNode(vnode.ssContent),  
 ssFallback: vnode.ssFallback && cloneVNode(vnode.ssFallback),  
 el: vnode.el,  
 anchor: vnode.anchor,  
 ctx: vnode.ctx,  
 ce: vnode.ce  
 };  
 if (transition && cloneTransition) {  
 setTransitionHooks(  
 cloned,  
 transition.clone(cloned)  
 );  
 }  
 return cloned;  
}  
function deepCloneVNode(vnode) {  
 const cloned = cloneVNode(vnode);  
 if (isArray(vnode.children)) {  
 cloned.children = vnode.children.map(deepCloneVNode);  
 }  
 return cloned;  
}  
function createTextVNode(text = " ", flag = 0) {  
 return createVNode(Text, null, text, flag);  
}  
function createStaticVNode(content, numberOfNodes) {  
 const vnode = createVNode(Static, null, content);  
 vnode.staticCount = numberOfNodes;  
 return vnode;  
}  
function createCommentVNode(text = "", asBlock = false) {  
 return asBlock ? (openBlock(), createBlock(Comment, null, text)) : createVNode(Comment, null, text);  
}  
function normalizeVNode(child) {  
 if (child == null || typeof child === "boolean") {  
 return createVNode(Comment);  
 } else if (isArray(child)) {  
 return createVNode(  
 Fragment,  
 null,  
 child.slice()  
 );  
 } else if (typeof child === "object") {  
 return cloneIfMounted(child);  
 } else {  
 return createVNode(Text, null, String(child));  
 }  
}  
function cloneIfMounted(child) {  
 return child.el === null && child.patchFlag !== -1 || child.memo ? child : cloneVNode(child);  
}  
function normalizeChildren(vnode, children) {  
 let type = 0;  
 const { shapeFlag } = vnode;  
 if (children == null) {  
 children = null;  
 } else if (isArray(children)) {  
 type = 16;  
 } else if (typeof children === "object") {  
 if (shapeFlag & (1 | 64)) {  
 const slot = children.default;  
 if (slot) {  
 slot.\_c && (slot.\_d = false);  
 normalizeChildren(vnode, slot());  
 slot.\_c && (slot.\_d = true);  
 }  
 return;  
 } else {  
 type = 32;  
 const slotFlag = children.\_;  
 if (!slotFlag && !isInternalObject(children)) {  
 children.\_ctx = currentRenderingInstance;  
 } else if (slotFlag === 3 && currentRenderingInstance) {  
 if (currentRenderingInstance.slots.\_ === 1) {  
 children.\_ = 1;  
 } else {  
 children.\_ = 2;  
 vnode.patchFlag |= 1024;  
 }  
 }  
 }  
 } else if (isFunction(children)) {  
 children = { default: children, \_ctx: currentRenderingInstance };  
 type = 32;  
 } else {  
 children = String(children);  
 if (shapeFlag & 64) {  
 type = 16;  
 children = [createTextVNode(children)];  
 } else {  
 type = 8;  
 }  
 }  
 vnode.children = children;  
 vnode.shapeFlag |= type;  
}  
function mergeProps(...args) {  
 const ret = {};  
 for (let i = 0; i < args.length; i++) {  
 const toMerge = args[i];  
 for (const key in toMerge) {  
 if (key === "class") {  
 if (ret.class !== toMerge.class) {  
 ret.class = normalizeClass([ret.class, toMerge.class]);  
 }  
 } else if (key === "style") {  
 ret.style = normalizeStyle([ret.style, toMerge.style]);  
 } else if (isOn(key)) {  
 const existing = ret[key];  
 const incoming = toMerge[key];  
  
 ret[key] = existing ? [].concat(existing, incoming) : incoming;  
 }  
 } else if (key !== "") {  
 ret[key] = toMerge[key];  
 }  
 }  
 }  
 return ret;  
}  
function invokeVNodeHook(hook, instance, vnode, prevVNode = null) {  
 callWithAsyncErrorHandling(hook, instance, 7, [  
 vnode,  
 prevVNode  
 ]);  
}  
var emptyAppContext = createAppContext();  
var uid = 0;  
function createComponentInstance(vnode, parent, suspense) {  
 const type = vnode.type;  
 const appContext = (parent ? parent.appContext : vnode.appContext) || emptyAppContext;  
 const instance = {  
 uid: uid++,  
 vnode,  
 type,  
 parent,  
 appContext,  
 root: null,  
 next: null,  
 subTree: null,  
 effect: null,  
 update: null,  
 scope: new EffectScope(  
 true  
 ),  
 render: null,  
 proxy: null,  
 exposed: null,  
 exposeProxy: null,  
 withProxy: null,  
 provides: parent ? parent.provides : Object.create(appContext.provides),  
 accessCache: null,  
 renderCache: [],  
 components: null,  
 directives: null,  
 propsOptions: normalizePropsOptions(type, appContext),  
 emitsOptions: normalizeEmitsOptions(type, appContext),  
 emit: null,  
 emitted: null,  
 propsDefaults: EMPTY\_OBJ,  
 inheritAttrs: type.inheritAttrs,  
 ctx: EMPTY\_OBJ,  
 data: EMPTY\_OBJ,  
 props: EMPTY\_OBJ,  
 attrs: EMPTY\_OBJ,  
 slots: EMPTY\_OBJ,  
 refs: EMPTY\_OBJ,  
 setupState: EMPTY\_OBJ,  
 setupContext: null,  
 suspense,  
 suspenseId: suspense ? suspense.pendingId : 0,  
 asyncDep: null,  
 asyncResolved: false,  
 isMounted: false,  
 isUnmounted: false,  
 isDeactivated: false,  
 bc: null,  
 c: null,  
 bm: null,  
 m: null,  
 bu: null,  
 u: null,  
 um: null,  
 bum: null,  
 da: null,  
 a: null,  
 rtg: null,  
 rtc: null,  
 ec: null,  
 sp: null  
 };  
 if (true) {  
 instance.ctx = createDevRenderContext(instance);  
 } else {  
 instance.ctx = { \_: instance };  
 }  
 instance.root = parent ? parent.root : instance;  
 instance.emit = emit.bind(null, instance);  
 if (vnode.ce) {  
 vnode.ce(instance);  
 }  
 return instance;  
}  
var currentInstance = null;  
var getCurrentInstance = () => currentInstance || currentRenderingInstance;  
var internalSetCurrentInstance;  
var setInSSRSetupState;  
{  
 const g = getGlobalThis();  
 const registerGlobalSetter = (key, setter) => {  
 let setters;  
 if (!(setters = g[key])) setters = g[key] = [];  
 setters.push(setter);  
 return (v) => {  
 if (setters.length > 1) setters.forEach((set2) => set2(v));  
 else setters[0](v);  
 };  
 };  
 internalSetCurrentInstance = registerGlobalSetter(  
 `\_\_VUE\_INSTANCE\_SETTERS\_\_`,  
 (v) => currentInstance = v  
 );  
 setInSSRSetupState = registerGlobalSetter(  
 `\_\_VUE\_SSR\_SETTERS\_\_`,  
 (v) => isInSSRComponentSetup = v  
 );  
}  
var setCurrentInstance = (instance) => {  
 const prev = currentInstance;  
 internalSetCurrentInstance(instance);  
 instance.scope.on();  
 return () => {  
 instance.scope.off();  
 internalSetCurrentInstance(prev);  
 };  
};  
var unsetCurrentInstance = () => {  
 currentInstance && currentInstance.scope.off();  
 internalSetCurrentInstance(null);  
};  
var isBuiltInTag = makeMap("slot,component");  
function validateComponentName(name, { isNativeTag }) {  
 if (isBuiltInTag(name) || isNativeTag(name)) {  
 warn$1(  
 "Do not use built-in or reserved HTML elements as component id: " + name  
 );  
 }  
}  
function isStatefulComponent(instance) {  
 return instance.vnode.shapeFlag & 4;  
}  
var isInSSRComponentSetup = false;  
function setupComponent(instance, isSSR = false, optimized = false) {  
 isSSR && setInSSRSetupState(isSSR);  
 const { props, children } = instance.vnode;  
 const isStateful = isStatefulComponent(instance);  
 initProps(instance, props, isStateful, isSSR);  
 initSlots(instance, children, optimized);  
 const setupResult = isStateful ? setupStatefulComponent(instance, isSSR) : void 0;  
 isSSR && setInSSRSetupState(false);  
 return setupResult;  
}  
function setupStatefulComponent(instance, isSSR) {  
 var \_a;  
 const Component = instance.type;  
 if (true) {  
 if (Component.name) {  
 validateComponentName(Component.name, instance.appContext.config);  
 }  
 if (Component.components) {  
 const names = Object.keys(Component.components);  
 for (let i = 0; i < names.length; i++) {  
 validateComponentName(names[i], instance.appContext.config);  
 }  
 }  
 if (Component.directives) {  
 const names = Object.keys(Component.directives);  
 for (let i = 0; i < names.length; i++) {  
 validateDirectiveName(names[i]);  
 }  
 }  
 if (Component.compilerOptions && isRuntimeOnly()) {  
 warn$1(  
 `"compilerOptions" is only supported when using a build of Vue that includes the runtime compiler. Since you are using a runtime-only build, the options should be passed via your build tool config instead.`  
 );  
 }  
 }  
 instance.accessCache = /\* @\_\_PURE\_\_ \*/ Object.create(null);  
 instance.proxy = new Proxy(instance.ctx, PublicInstanceProxyHandlers);  
 if (true) {  
 exposePropsOnRenderContext(instance);  
 }  
 const { setup } = Component;  
 if (setup) {  
 const setupContext = instance.setupContext = setup.length > 1 ? createSetupContext(instance) : null;  
 const reset = setCurrentInstance(instance);  
 pauseTracking();  
 const setupResult = callWithErrorHandling(  
 setup,  
 instance,  
 0,  
 [  
 true ? shallowReadonly(instance.props) : instance.props,  
 setupContext  
 ]  
 );  
 resetTracking();  
 reset();  
 if (isPromise(setupResult)) {  
 setupResult.then(unsetCurrentInstance, unsetCurrentInstance);  
 if (isSSR) {  
 return setupResult.then((resolvedResult) => {  
 handleSetupResult(instance, resolvedResult, isSSR);  
 }).catch((e) => {  
 handleError(e, instance, 0);  
 });  
 } else {  
 instance.asyncDep = setupResult;  
 if (!instance.suspense) {  
 const name = (\_a = Component.name) != null ? \_a : "Anonymous";  
 warn$1(  
 `Component <${name}>: setup function returned a promise, but no <Suspense> boundary was found in the parent component tree. A component with async setup() must be nested in a <Suspense> in order to be rendered.`  
 );  
 }  
 }  
 } else {  
 handleSetupResult(instance, setupResult, isSSR);  
 }  
 } else {  
 finishComponentSetup(instance, isSSR);  
 }  
}  
function handleSetupResult(instance, setupResult, isSSR) {  
 if (isFunction(setupResult)) {  
 if (instance.type.\_\_ssrInlineRender) {  
 instance.ssrRender = setupResult;  
 } else {  
 instance.render = setupResult;  
 }  
 } else if (isObject(setupResult)) {  
 if (isVNode(setupResult)) {  
 warn$1(  
 `setup() should not return VNodes directly - return a render function instead.`  
 );  
 }  
 if (true) {  
 instance.devtoolsRawSetupState = setupResult;  
 }  
 instance.setupState = proxyRefs(setupResult);  
 if (true) {  
 exposeSetupStateOnRenderContext(instance);  
 }  
 } else if (setupResult !== void 0) {  
 warn$1(  
 `setup() should return an object. Received: ${setupResult === null ? "null" : typeof setupResult}`  
 );  
 }  
 finishComponentSetup(instance, isSSR);  
}  
var compile;  
var installWithProxy;  
function registerRuntimeCompiler(\_compile) {  
 compile = \_compile;  
 installWithProxy = (i) => {  
 if (i.render.\_rc) {  
 i.withProxy = new Proxy(i.ctx, RuntimeCompiledPublicInstanceProxyHandlers);  
 }  
 };  
}  
var isRuntimeOnly = () => !compile;  
function finishComponentSetup(instance, isSSR, skipOptions) {  
 const Component = instance.type;  
 if (!instance.render) {  
 if (!isSSR && compile && !Component.render) {  
 const template = Component.template || resolveMergedOptions(instance).template;  
 if (template) {  
 if (true) {  
 startMeasure(instance, `compile`);  
 }  
 const { isCustomElement, compilerOptions } = instance.appContext.config;  
 const { delimiters, compilerOptions: componentCompilerOptions } = Component;  
 const finalCompilerOptions = extend(  
 extend(  
 {  
 isCustomElement,  
 delimiters  
 },  
 compilerOptions  
 ),  
 componentCompilerOptions  
 );  
 Component.render = compile(template, finalCompilerOptions);  
 if (true) {  
 endMeasure(instance, `compile`);  
 }  
 }  
 }  
 instance.render = Component.render || NOOP;  
 if (installWithProxy) {  
 installWithProxy(instance);  
 }  
 }  
 if (\_\_VUE\_OPTIONS\_API\_\_ && true) {  
 const reset = setCurrentInstance(instance);  
 pauseTracking();  
 try {  
 applyOptions(instance);  
 } finally {  
 resetTracking();  
 reset();  
 }  
 }  
 if (!Component.render && instance.render === NOOP && !isSSR) {  
 if (!compile && Component.template) {  
 warn$1(  
 `Component provided template option but runtime compilation is not supported in this build of Vue. Configure your bundler to alias "vue" to "vue/dist/vue.esm-bundler.js".`  
 );  
 } else {  
 warn$1(`Component is missing template or render function: `, Component);  
 }  
 }  
}  
var attrsProxyHandlers = true ? {  
 get(target, key) {  
 markAttrsAccessed();  
 track(target, "get", "");  
 return target[key];  
 },  
 set() {  
 warn$1(`setupContext.attrs is readonly.`);  
 return false;  
 },  
 deleteProperty() {  
 warn$1(`setupContext.attrs is readonly.`);  
 return false;  
 }  
} : {  
 get(target, key) {  
 track(target, "get", "");  
 return target[key];  
 }  
};  
function getSlotsProxy(instance) {  
 return new Proxy(instance.slots, {  
 get(target, key) {  
 track(instance, "get", "$slots");  
 return target[key];  
 }  
 });  
}  
function createSetupContext(instance) {  
 const expose = (exposed) => {  
 if (true) {  
 if (instance.exposed) {  
 warn$1(`expose() should be called only once per setup().`);  
 }  
 if (exposed != null) {  
 let exposedType = typeof exposed;  
 if (exposedType === "object") {  
 if (isArray(exposed)) {  
 exposedType = "array";  
 } else if (isRef2(exposed)) {  
 exposedType = "ref";  
 }  
 }  
 if (exposedType !== "object") {  
 warn$1(  
 `expose() should be passed a plain object, received ${exposedType}.`  
 );  
 }  
 }  
 }  
 instance.exposed = exposed || {};  
 };  
 if (true) {  
 let attrsProxy;  
 let slotsProxy;  
 return Object.freeze({  
 get attrs() {  
 return attrsProxy || (attrsProxy = new Proxy(instance.attrs, attrsProxyHandlers));  
 },  
 get slots() {  
 return slotsProxy || (slotsProxy = getSlotsProxy(instance));  
 },  
 get emit() {  
 return (event, ...args) => instance.emit(event, ...args);  
 },  
 expose  
 });  
 } else {  
 return {  
 attrs: new Proxy(instance.attrs, attrsProxyHandlers),  
 slots: instance.slots,  
 emit: instance.emit,  
 expose  
 };  
 }  
}  
function getComponentPublicInstance(instance) {  
 if (instance.exposed) {  
 return instance.exposeProxy || (instance.exposeProxy = new Proxy(proxyRefs(markRaw(instance.exposed)), {  
 get(target, key) {  
 if (key in target) {  
 return target[key];  
 } else if (key in publicPropertiesMap) {  
 return publicPropertiesMap[key](instance);  
 }  
 },  
 has(target, key) {  
 return key in target || key in publicPropertiesMap;  
 }  
 }));  
 } else {  
 return instance.proxy;  
 }  
}  
var classifyRE = /(?:^|[-\_])(\w)/g;  
var classify = (str) => str.replace(classifyRE, (c) => c.toUpperCase()).replace(/[-\_]/g, "");  
function getComponentName(Component, includeInferred = true) {  
 return isFunction(Component) ? Component.displayName || Component.name : Component.name || includeInferred && Component.\_\_name;  
}  
function formatComponentName(instance, Component, isRoot = false) {  
 let name = getComponentName(Component);  
 if (!name && Component.\_\_file) {  
 const match = Component.\_\_file.match(/([^/\\]+)\.\w+$/);  
 if (match) {  
 name = match[1];  
 }  
 }  
 if (!name && instance && instance.parent) {  
 const inferFromRegistry = (registry) => {  
 for (const key in registry) {  
 if (registry[key] === Component) {  
 return key;  
 }  
 }  
 };  
 name = inferFromRegistry(  
 instance.components || instance.parent.type.components  
 ) || inferFromRegistry(instance.appContext.components);  
 }  
 return name ? classify(name) : isRoot ? `App` : `Anonymous`;  
}  
function isClassComponent(value) {  
 return isFunction(value) && "\_\_vccOpts" in value;  
}  
var computed2 = (getterOrOptions, debugOptions) => {  
 const c = computed(getterOrOptions, debugOptions, isInSSRComponentSetup);  
 if (true) {  
 const i = getCurrentInstance();  
 if (i && i.appContext.config.warnRecursiveComputed) {  
 c.\_warnRecursive = true;  
 }  
 }  
 return c;  
};  
function h(type, propsOrChildren, children) {  
 const l = arguments.length;  
 if (l === 2) {  
 if (isObject(propsOrChildren) && !isArray(propsOrChildren)) {  
 if (isVNode(propsOrChildren)) {  
 return createVNode(type, null, [propsOrChildren]);  
 }  
 return createVNode(type, propsOrChildren);  
 } else {  
 return createVNode(type, null, propsOrChildren);  
 }  
 } else {  
 if (l > 3) {  
 children = Array.prototype.slice.call(arguments, 2);  
 } else if (l === 3 && isVNode(children)) {  
 children = [children];  
 }  
 return createVNode(type, propsOrChildren, children);  
 }  
}  
function initCustomFormatter() {  
 if (typeof window === "undefined") {  
 return;  
 }  
 const vueStyle = { style: "color:#3ba776" };  
 const numberStyle = { style: "color:#1677ff" };  
 const stringStyle = { style: "color:#f5222d" };  
 const keywordStyle = { style: "color:#eb2f96" };  
 const formatter = {  
 \_\_vue\_custom\_formatter: true,  
 header(obj) {  
 if (!isObject(obj)) {  
 return null;  
 }  
 if (obj.\_\_isVue) {  
 return ["div", vueStyle, `VueInstance`];  
 } else if (isRef2(obj)) {  
 return [  
 "div",  
 {},  
 ["span", vueStyle, genRefFlag(obj)],  
 "<",  
 formatValue(obj.value),  
 `>`  
 ];  
 } else if (isReactive(obj)) {  
 return [  
 "div",  
 {},  
 ["span", vueStyle, isShallow(obj) ? "ShallowReactive" : "Reactive"],  
 "<",  
 formatValue(obj),  
 `>${isReadonly(obj) ? ` (readonly)` : ``}`  
 ];  
 } else if (isReadonly(obj)) {  
 return [  
 "div",  
 {},  
 ["span", vueStyle, isShallow(obj) ? "ShallowReadonly" : "Readonly"],  
 "<",  
 formatValue(obj),  
 ">"  
 ];  
 }  
 return null;  
 },  
 hasBody(obj) {  
 return obj && obj.\_\_isVue;  
 },  
 body(obj) {  
 if (obj && obj.\_\_isVue) {  
 return [  
 "div",  
 {},  
 ...formatInstance(obj.$)  
 ];  
 }  
 }  
 };  
 function formatInstance(instance) {  
 const blocks = [];  
 if (instance.type.props && instance.props) {  
 blocks.push(createInstanceBlock("props", toRaw(instance.props)));  
 }  
 if (instance.setupState !== EMPTY\_OBJ) {  
 blocks.push(createInstanceBlock("setup", instance.setupState));  
 }  
 if (instance.data !== EMPTY\_OBJ) {  
 blocks.push(createInstanceBlock("data", toRaw(instance.data)));  
 }  
 const computed3 = extractKeys(instance, "computed");  
 if (computed3) {  
 blocks.push(createInstanceBlock("computed", computed3));  
 }  
 const injected = extractKeys(instance, "inject");  
 if (injected) {  
 blocks.push(createInstanceBlock("injected", injected));  
 }  
 blocks.push([  
 "div",  
 {},  
 [  
 "span",  
 {  
 style: keywordStyle.style + ";opacity:0.66"  
 },  
 "$ (internal): "  
 ],  
 ["object", { object: instance }]  
 ]);  
 return blocks;  
 }  
 function createInstanceBlock(type, target) {  
 target = extend({}, target);  
 if (!Object.keys(target).length) {  
 return ["span", {}];  
 }  
 return [  
 "div",  
 { style: "line-height:1.25em;margin-bottom:0.6em" },  
 [  
 "div",  
 {  
 style: "color:#476582"  
 },  
 type  
 ],  
 [  
 "div",  
 {  
 style: "padding-left:1.25em"  
 },  
 ...Object.keys(target).map((key) => {  
 return [  
 "div",  
 {},  
 ["span", keywordStyle, key + ": "],  
 formatValue(target[key], false)  
 ];  
 })  
 ]  
 ];  
 }  
 function formatValue(v, asRaw = true) {  
 if (typeof v === "number") {  
 return ["span", numberStyle, v];  
 } else if (typeof v === "string") {  
 return ["span", stringStyle, JSON.stringify(v)];  
 } else if (typeof v === "boolean") {  
 return ["span", keywordStyle, v];  
 } else if (isObject(v)) {  
 return ["object", { object: asRaw ? toRaw(v) : v }];  
 } else {  
 return ["span", stringStyle, String(v)];  
 }  
 }  
 function extractKeys(instance, type) {  
 const Comp = instance.type;  
 if (isFunction(Comp)) {  
 return;  
 }  
 const extracted = {};  
 for (const key in instance.ctx) {  
 if (isKeyOfType(Comp, key, type)) {  
 extracted[key] = instance.ctx[key];  
 }  
 }  
 return extracted;  
 }  
 function isKeyOfType(Comp, key, type) {  
 const opts = Comp[type];  
 if (isArray(opts) && opts.includes(key) || isObject(opts) && key in opts) {  
 return true;  
 }  
 if (Comp.extends && isKeyOfType(Comp.extends, key, type)) {  
 return true;  
 }  
 if (Comp.mixins && Comp.mixins.some((m) => isKeyOfType(m, key, type))) {  
 return true;  
 }  
 }  
 function genRefFlag(v) {  
 if (isShallow(v)) {  
 return `ShallowRef`;  
 }  
 if (v.effect) {  
 return `ComputedRef`;  
 }  
 return `Ref`;  
 }  
 if (window.devtoolsFormatters) {  
 window.devtoolsFormatters.push(formatter);  
 } else {  
 window.devtoolsFormatters = [formatter];  
 }  
}  
function withMemo(memo, render2, cache, index) {  
 const cached = cache[index];  
 if (cached && isMemoSame(cached, memo)) {  
 return cached;  
 }  
 const ret = render2();  
 ret.memo = memo.slice();  
 ret.cacheIndex = index;  
 return cache[index] = ret;  
}  
function isMemoSame(cached, memo) {  
 const prev = cached.memo;  
 if (prev.length != memo.length) {  
 return false;  
 }  
 for (let i = 0; i < prev.length; i++) {  
 if (hasChanged(prev[i], memo[i])) {  
 return false;  
 }  
 }  
 if (isBlockTreeEnabled > 0 && currentBlock) {  
 currentBlock.push(cached);  
 }  
 return true;  
}  
var version = "3.4.33";  
var warn2 = true ? warn$1 : NOOP;  
var ErrorTypeStrings = ErrorTypeStrings$1;  
var devtools = true ? devtools$1 : void 0;  
var setDevtoolsHook = true ? setDevtoolsHook$1 : NOOP;  
var \_ssrUtils = {  
 createComponentInstance,  
 setupComponent,  
 renderComponentRoot,  
 setCurrentRenderingInstance,  
 isVNode,  
 normalizeVNode,  
 getComponentPublicInstance  
};  
var ssrUtils = \_ssrUtils;  
var resolveFilter = null;  
var compatUtils = null;  
var DeprecationTypes = null;  
var svgNS = "http://www.w3.org/2000/svg";  
var mathmlNS = "http://www.w3.org/1998/Math/MathML";  
var doc = typeof document !== "undefined" ? document : null;  
var templateContainer = doc && doc.createElement("template");  
var nodeOps = {  
 insert: (child, parent, anchor) => {  
 parent.insertBefore(child, anchor || null);  
 },  
 remove: (child) => {  
 const parent = child.parentNode;  
 if (parent) {  
 parent.removeChild(child);  
 }  
 },  
 createElement: (tag, namespace, is, props) => {  
 const el = namespace === "svg" ? doc.createElementNS(svgNS, tag) : namespace === "mathml" ? doc.createElementNS(mathmlNS, tag) : is ? doc.createElement(tag, { is }) : doc.createElement(tag);  
 if (tag === "select" && props && props.multiple != null) {  
 el.setAttribute("multiple", props.multiple);  
 }  
 return el;  
 },  
 createText: (text) => doc.createTextNode(text),  
 createComment: (text) => doc.createComment(text),  
 setText: (node, text) => {  
 node.nodeValue = text;  
 },  
 setElementText: (el, text) => {  
 el.textContent = text;  
 },  
 parentNode: (node) => node.parentNode,  
 nextSibling: (node) => node.nextSibling,  
 querySelector: (selector) => doc.querySelector(selector),  
 setScopeId(el, id) {  
 el.setAttribute(id, "");  
 },  
 insertStaticContent(content, parent, anchor, namespace, start, end) {  
 const before = anchor ? anchor.previousSibling : parent.lastChild;  
 if (start && (start === end || start.nextSibling)) {  
 while (true) {  
 parent.insertBefore(start.cloneNode(true), anchor);  
 if (start === end || !(start = start.nextSibling)) break;  
 }  
 } else {  
 templateContainer.innerHTML = namespace === "svg" ? `<svg>${content}</svg>` : namespace === "mathml" ? `<math>${content}</math>` : content;  
 const template = templateContainer.content;  
 if (namespace === "svg" || namespace === "mathml") {  
 const wrapper = template.firstChild;  
 while (wrapper.firstChild) {  
 template.appendChild(wrapper.firstChild);  
 }  
 template.removeChild(wrapper);  
 }  
 parent.insertBefore(template, anchor);  
 }  
 return [  
 before ? before.nextSibling : parent.firstChild,  
 anchor ? anchor.previousSibling : parent.lastChild  
 ];  
 }  
};  
var TRANSITION = "transition";  
var ANIMATION = "animation";  
var vtcKey = Symbol("\_vtc");  
var Transition = (props, { slots }) => h(BaseTransition, resolveTransitionProps(props), slots);  
Transition.displayName = "Transition";  
var DOMTransitionPropsValidators = {  
 name: String,  
 type: String,  
 css: {  
 type: Boolean,  
 default: true  
 },  
 duration: [String, Number, Object],  
 enterFromClass: String,  
 enterActiveClass: String,  
 enterToClass: String,  
 appearFromClass: String,  
 appearActiveClass: String,  
 appearToClass: String,  
 leaveFromClass: String,  
 leaveActiveClass: String,  
 leaveToClass: String  
};  
var TransitionPropsValidators = Transition.props = extend(  
 {},  
 BaseTransitionPropsValidators,  
 DOMTransitionPropsValidators  
);  
var callHook2 = (hook, args = []) => {  
 if (isArray(hook)) {  
 hook.forEach((h2) => h2(...args));  
 } else if (hook) {  
 hook(...args);  
 }  
};  
var hasExplicitCallback = (hook) => {  
 return hook ? isArray(hook) ? hook.some((h2) => h2.length > 1) : hook.length > 1 : false;  
};  
function resolveTransitionProps(rawProps) {  
 const baseProps = {};  
 for (const key in rawProps) {  
 if (!(key in DOMTransitionPropsValidators)) {  
 baseProps[key] = rawProps[key];  
 }  
 }  
 if (rawProps.css === false) {  
 return baseProps;  
 }  
 const {  
 name = "v",  
 type,  
 duration,  
 enterFromClass = `${name}-enter-from`,  
 enterActiveClass = `${name}-enter-active`,  
 enterToClass = `${name}-enter-to`,  
 appearFromClass = enterFromClass,  
 appearActiveClass = enterActiveClass,  
 appearToClass = enterToClass,  
 leaveFromClass = `${name}-leave-from`,  
 leaveActiveClass = `${name}-leave-active`,  
 leaveToClass = `${name}-leave-to`  
 } = rawProps;  
 const durations = normalizeDuration(duration);  
 const enterDuration = durations && durations[0];  
 const leaveDuration = durations && durations[1];  
 const {  
 onBeforeEnter,  
 onEnter,  
 onEnterCancelled,  
 onLeave,  
 onLeaveCancelled,  
 onBeforeAppear = onBeforeEnter,  
 onAppear = onEnter,  
 onAppearCancelled = onEnterCancelled  
 } = baseProps;  
 const finishEnter = (el, isAppear, done) => {  
 removeTransitionClass(el, isAppear ? appearToClass : enterToClass);  
 removeTransitionClass(el, isAppear ? appearActiveClass : enterActiveClass);  
 done && done();  
 };  
 const finishLeave = (el, done) => {  
 el.\_isLeaving = false;  
 removeTransitionClass(el, leaveFromClass);  
 removeTransitionClass(el, leaveToClass);  
 removeTransitionClass(el, leaveActiveClass);  
 done && done();  
 };  
 const makeEnterHook = (isAppear) => {  
 return (el, done) => {  
 const hook = isAppear ? onAppear : onEnter;  
 const resolve2 = () => finishEnter(el, isAppear, done);  
 callHook2(hook, [el, resolve2]);  
 nextFrame(() => {  
 removeTransitionClass(el, isAppear ? appearFromClass : enterFromClass);  
 addTransitionClass(el, isAppear ? appearToClass : enterToClass);  
 if (!hasExplicitCallback(hook)) {  
 whenTransitionEnds(el, type, enterDuration, resolve2);  
 }  
 });  
 };  
 };  
 return extend(baseProps, {  
 onBeforeEnter(el) {  
 callHook2(onBeforeEnter, [el]);  
 addTransitionClass(el, enterFromClass);  
 addTransitionClass(el, enterActiveClass);  
 },  
 onBeforeAppear(el) {  
 callHook2(onBeforeAppear, [el]);  
 addTransitionClass(el, appearFromClass);  
 addTransitionClass(el, appearActiveClass);  
 },  
 onEnter: makeEnterHook(false),  
 onAppear: makeEnterHook(true),  
 onLeave(el, done) {  
 el.\_isLeaving = true;  
 const resolve2 = () => finishLeave(el, done);  
 addTransitionClass(el, leaveFromClass);  
 addTransitionClass(el, leaveActiveClass);  
 forceReflow();  
 nextFrame(() => {  
 if (!el.\_isLeaving) {  
 return;  
 }  
 removeTransitionClass(el, leaveFromClass);  
 addTransitionClass(el, leaveToClass);  
 if (!hasExplicitCallback(onLeave)) {  
 whenTransitionEnds(el, type, leaveDuration, resolve2);  
 }  
 });  
 callHook2(onLeave, [el, resolve2]);  
 },  
 onEnterCancelled(el) {  
 finishEnter(el, false);  
 callHook2(onEnterCancelled, [el]);  
 },  
 onAppearCancelled(el) {  
 finishEnter(el, true);  
 callHook2(onAppearCancelled, [el]);  
 },  
 onLeaveCancelled(el) {  
 finishLeave(el);  
 callHook2(onLeaveCancelled, [el]);  
 }  
 });  
}  
function normalizeDuration(duration) {  
 if (duration == null) {  
 return null;  
 } else if (isObject(duration)) {  
 return [NumberOf(duration.enter), NumberOf(duration.leave)];  
 } else {  
 const n = NumberOf(duration);  
 return [n, n];  
 }  
}  
function NumberOf(val) {  
 const res = toNumber(val);  
 if (true) {  
 assertNumber(res, "<transition> explicit duration");  
 }  
 return res;  
}  
function addTransitionClass(el, cls) {  
 cls.split(/\s+/).forEach((c) => c && el.classList.add(c));  
 (el[vtcKey] || (el[vtcKey] = /\* @\_\_PURE\_\_ \*/ new Set())).add(cls);  
}  
function removeTransitionClass(el, cls) {  
 cls.split(/\s+/).forEach((c) => c && el.classList.remove(c));  
 const \_vtc = el[vtcKey];  
 if (\_vtc) {  
 \_vtc.delete(cls);  
 if (!\_vtc.size) {  
 el[vtcKey] = void 0;  
 }  
 }  
}  
function nextFrame(cb) {  
 requestAnimationFrame(() => {  
 requestAnimationFrame(cb);  
 });  
}  
var endId = 0;  
function whenTransitionEnds(el, expectedType, explicitTimeout, resolve2) {  
 const id = el.\_endId = ++endId;  
 const resolveIfNotStale = () => {  
 if (id === el.\_endId) {  
 resolve2();  
 }  
 };  
 if (explicitTimeout) {  
 return setTimeout(resolveIfNotStale, explicitTimeout);  
 }  
 const { type, timeout, propCount } = getTransitionInfo(el, expectedType);  
 if (!type) {  
 return resolve2();  
 }  
 const endEvent = type + "end";  
 let ended = 0;  
 const end = () => {  
 el.removeEventListener(endEvent, onEnd);  
 resolveIfNotStale();  
 };  
 const onEnd = (e) => {  
 if (e.target === el && ++ended >= propCount) {  
 end();  
 }  
 };  
 setTimeout(() => {  
 if (ended < propCount) {  
 end();  
 }  
 }, timeout + 1);  
 el.addEventListener(endEvent, onEnd);  
}  
function getTransitionInfo(el, expectedType) {  
 const styles = window.getComputedStyle(el);  
 const getStyleProperties = (key) => (styles[key] || "").split(", ");  
 const transitionDelays = getStyleProperties(`${TRANSITION}Delay`);  
 const transitionDurations = getStyleProperties(`${TRANSITION}Duration`);  
 const transitionTimeout = getTimeout(transitionDelays, transitionDurations);  
 const animationDelays = getStyleProperties(`${ANIMATION}Delay`);  
 const animationDurations = getStyleProperties(`${ANIMATION}Duration`);  
 const animationTimeout = getTimeout(animationDelays, animationDurations);  
 let type = null;  
 let timeout = 0;  
 let propCount = 0;  
 if (expectedType === TRANSITION) {  
 if (transitionTimeout > 0) {  
 type = TRANSITION;  
 timeout = transitionTimeout;  
 propCount = transitionDurations.length;  
 }  
 } else if (expectedType === ANIMATION) {  
 if (animationTimeout > 0) {  
 type = ANIMATION;  
 timeout = animationTimeout;  
 propCount = animationDurations.length;  
 }  
 } else {  
 timeout = Math.max(transitionTimeout, animationTimeout);  
 type = timeout > 0 ? transitionTimeout > animationTimeout ? TRANSITION : ANIMATION : null;  
 propCount = type ? type === TRANSITION ? transitionDurations.length : animationDurations.length : 0;  
 }  
 const hasTransform = type === TRANSITION && /\b(transform|all)(,|$)/.test(  
 getStyleProperties(`${TRANSITION}Property`).toString()  
 );  
 return {  
 type,  
 timeout,  
 propCount,  
 hasTransform  
 };  
}  
function getTimeout(delays, durations) {  
 while (delays.length < durations.length) {  
 delays = delays.concat(delays);  
 }  
 return Math.max(...durations.map((d, i) => toMs(d) + toMs(delays[i])));  
}  
function toMs(s) {  
 if (s === "auto") return 0;  
 return Number(s.slice(0, -1).replace(",", ".")) \* 1e3;  
}  
function forceReflow() {  
 return document.body.offsetHeight;  
}  
function patchClass(el, value, isSVG) {  
 const transitionClasses = el[vtcKey];  
 if (transitionClasses) {  
 value = (value ? [value, ...transitionClasses] : [...transitionClasses]).join(" ");  
 }  
 if (value == null) {  
 el.removeAttribute("class");  
 } else if (isSVG) {  
 el.setAttribute("class", value);  
 } else {  
 el.className = value;  
 }  
}  
var vShowOriginalDisplay = Symbol("\_vod");  
var vShowHidden = Symbol("\_vsh");  
var vShow = {  
 beforeMount(el, { value }, { transition }) {  
 el[vShowOriginalDisplay] = el.style.display === "none" ? "" : el.style.display;  
 if (transition && value) {  
 transition.beforeEnter(el);  
 } else {  
 setDisplay(el, value);  
 }  
 },  
 mounted(el, { value }, { transition }) {  
 if (transition && value) {  
 transition.enter(el);  
 }  
 },  
 updated(el, { value, oldValue }, { transition }) {  
 if (!value === !oldValue) return;  
 if (transition) {  
 if (value) {  
 transition.beforeEnter(el);  
 setDisplay(el, true);  
 transition.enter(el);  
 } else {  
 transition.leave(el, () => {  
 setDisplay(el, false);  
 });  
 }  
 } else {  
 setDisplay(el, value);  
 }  
 },  
 beforeUnmount(el, { value }) {  
 setDisplay(el, value);  
 }  
};  
if (true) {  
 vShow.name = "show";  
}  
function setDisplay(el, value) {  
 el.style.display = value ? el[vShowOriginalDisplay] : "none";  
 el[vShowHidden] = !value;  
}  
function initVShowForSSR() {  
 vShow.getSSRProps = ({ value }) => {  
 if (!value) {  
 return { style: { display: "none" } };  
 }  
 };  
}  
var CSS\_VAR\_TEXT = Symbol(true ? "CSS\_VAR\_TEXT" : "");  
function useCssVars(getter) {  
 const instance = getCurrentInstance();  
 if (!instance) {  
 warn2(`useCssVars is called without current active component instance.`);  
 return;  
 }  
 const updateTeleports = instance.ut = (vars = getter(instance.proxy)) => {  
 Array.from(  
 document.querySelectorAll(`[data-v-owner="${instance.uid}"]`)  
 ).forEach((node) => setVarsOnNode(node, vars));  
 };  
 if (true) {  
 instance.getCssVars = () => getter(instance.proxy);  
 }  
 const setVars = () => {  
 const vars = getter(instance.proxy);  
 setVarsOnVNode(instance.subTree, vars);  
 updateTeleports(vars);  
 };  
 onMounted(() => {  
 watchPostEffect(setVars);  
 const ob = new MutationObserver(setVars);  
 ob.observe(instance.subTree.el.parentNode, { childList: true });  
 onUnmounted(() => ob.disconnect());  
 });  
}  
function setVarsOnVNode(vnode, vars) {  
 if (vnode.shapeFlag & 128) {  
 const suspense = vnode.suspense;  
 vnode = suspense.activeBranch;  
 if (suspense.pendingBranch && !suspense.isHydrating) {  
 suspense.effects.push(() => {  
 setVarsOnVNode(suspense.activeBranch, vars);  
 });  
 }  
 }  
 while (vnode.component) {  
 vnode = vnode.component.subTree;  
 }  
 if (vnode.shapeFlag & 1 && vnode.el) {  
 setVarsOnNode(vnode.el, vars);  
 } else if (vnode.type === Fragment) {  
 vnode.children.forEach((c) => setVarsOnVNode(c, vars));  
 } else if (vnode.type === Static) {  
 let { el, anchor } = vnode;  
 while (el) {  
 setVarsOnNode(el, vars);  
 if (el === anchor) break;  
 el = el.nextSibling;  
 }  
 }  
}  
function setVarsOnNode(el, vars) {  
 if (el.nodeType === 1) {  
 const style = el.style;  
 let cssText = "";  
 for (const key in vars) {  
 style.setProperty(`--${key}`, vars[key]);  
 cssText += `--${key}: ${vars[key]};`;  
 }  
 style[CSS\_VAR\_TEXT] = cssText;  
 }  
}  
var displayRE = /(^|;)\s\*display\s\*:/;  
function patchStyle(el, prev, next) {  
 const style = el.style;  
 const isCssString = isString(next);  
 let hasControlledDisplay = false;  
 if (next && !isCssString) {  
 if (prev) {  
 if (!isString(prev)) {  
 for (const key in prev) {  
 if (next[key] == null) {  
 setStyle(style, key, "");  
 }  
 }  
 } else {  
 for (const prevStyle of prev.split(";")) {  
 const key = prevStyle.slice(0, prevStyle.indexOf(":")).trim();  
 if (next[key] == null) {  
 setStyle(style, key, "");  
 }  
 }  
 }  
 }  
 for (const key in next) {  
 if (key === "display") {  
 hasControlledDisplay = true;  
 }  
 setStyle(style, key, next[key]);  
 }  
 } else {  
 if (isCssString) {  
 if (prev !== next) {  
 const cssVarText = style[CSS\_VAR\_TEXT];  
 if (cssVarText) {  
 next += ";" + cssVarText;  
 }  
 style.cssText = next;  
 hasControlledDisplay = displayRE.test(next);  
 }  
 } else if (prev) {  
 el.removeAttribute("style");  
 }  
 }  
 if (vShowOriginalDisplay in el) {  
 el[vShowOriginalDisplay] = hasControlledDisplay ? style.display : "";  
 if (el[vShowHidden]) {  
 style.display = "none";  
 }  
 }  
}  
var semicolonRE = /[^\\];\s\*$/;  
var importantRE = /\s\*!important$/;  
function setStyle(style, name, val) {  
 if (isArray(val)) {  
 val.forEach((v) => setStyle(style, name, v));  
 } else {  
 if (val == null) val = "";  
 if (true) {  
 if (semicolonRE.test(val)) {  
 warn2(  
 `Unexpected semicolon at the end of '${name}' style value: '${val}'`  
 );  
 }  
 }  
 if (name.startsWith("--")) {  
 style.setProperty(name, val);  
 } else {  
 const prefixed = autoPrefix(style, name);  
 if (importantRE.test(val)) {  
 style.setProperty(  
 hyphenate(prefixed),  
 val.replace(importantRE, ""),  
 "important"  
 );  
 } else {  
 style[prefixed] = val;  
 }  
 }  
 }  
}  
var prefixes = ["Webkit", "Moz", "ms"];  
var prefixCache = {};  
function autoPrefix(style, rawName) {  
 const cached = prefixCache[rawName];  
 if (cached) {  
 return cached;  
 }  
 let name = camelize(rawName);  
 if (name !== "filter" && name in style) {  
 return prefixCache[rawName] = name;  
 }  
 name = capitalize(name);  
 for (let i = 0; i < prefixes.length; i++) {  
 const prefixed = prefixes[i] + name;  
 if (prefixed in style) {  
 return prefixCache[rawName] = prefixed;  
 }  
 }  
 return rawName;  
}  
var xlinkNS = "http://www.w3.org/1999/xlink";  
function patchAttr(el, key, value, isSVG, instance, isBoolean2 = isSpecialBooleanAttr(key)) {  
 if (isSVG && key.startsWith("xlink:")) {  
 if (value == null) {  
 el.removeAttributeNS(xlinkNS, key.slice(6, key.length));  
 } else {  
 el.setAttributeNS(xlinkNS, key, value);  
 }  
 } else {  
 if (value == null || isBoolean2 && !includeBooleanAttr(value)) {  
 el.removeAttribute(key);  
 } else {  
 el.setAttribute(  
 key,  
 isBoolean2 ? "" : isSymbol(value) ? String(value) : value  
 );  
 }  
 }  
}  
function patchDOMProp(el, key, value, parentComponent) {  
 if (key === "innerHTML" || key === "textContent") {  
 if (value == null) return;  
 el[key] = value;  
 return;  
 }  
 const tag = el.tagName;  
 if (key === "value" && tag !== "PROGRESS" && // custom elements may use \_value internally  
 !tag.includes("-")) {  
 const oldValue = tag === "OPTION" ? el.getAttribute("value") || "" : el.value;  
 const newValue = value == null ? "" : String(value);  
 if (oldValue !== newValue || !("\_value" in el)) {  
 el.value = newValue;  
 }  
 if (value == null) {  
 el.removeAttribute(key);  
 }  
 el.\_value = value;  
 return;  
 }  
 let needRemove = false;  
 if (value === "" || value == null) {  
 const type = typeof el[key];  
 if (type === "boolean") {  
 value = includeBooleanAttr(value);  
 } else if (value == null && type === "string") {  
 value = "";  
 needRemove = true;  
 } else if (type === "number") {  
 value = 0;  
 needRemove = true;  
 }  
 }  
 try {  
 el[key] = value;  
 } catch (e) {  
 if (!needRemove) {  
 warn2(  
 `Failed setting prop "${key}" on <${tag.toLowerCase()}>: value ${value} is invalid.`,  
 e  
 );  
 }  
 }  
 needRemove && el.removeAttribute(key);  
}  
function addEventListener(el, event, handler, options) {  
 el.addEventListener(event, handler, options);  
}  
function removeEventListener(el, event, handler, options) {  
 el.removeEventListener(event, handler, options);  
}  
var veiKey = Symbol("\_vei");  
function patchEvent(el, rawName, prevValue, nextValue, instance = null) {  
 const invokers = el[veiKey] || (el[veiKey] = {});  
 const existingInvoker = invokers[rawName];  
 if (nextValue && existingInvoker) {  
 existingInvoker.value = true ? sanitizeEventValue(nextValue, rawName) : nextValue;  
 } else {  
 const [name, options] = parseName(rawName);  
 if (nextValue) {  
 const invoker = invokers[rawName] = createInvoker(  
 true ? sanitizeEventValue(nextValue, rawName) : nextValue,  
 instance  
 );  
 addEventListener(el, name, invoker, options);  
 } else if (existingInvoker) {  
 removeEventListener(el, name, existingInvoker, options);  
 invokers[rawName] = void 0;  
 }  
 }  
}  
var optionsModifierRE = /(?:Once|Passive|Capture)$/;  
function parseName(name) {  
 let options;  
 if (optionsModifierRE.test(name)) {  
 options = {};  
 let m;  
 while (m = name.match(optionsModifierRE)) {  
 name = name.slice(0, name.length - m[0].length);  
 options[m[0].toLowerCase()] = true;  
 }  
 }  
 const event = name[2] === ":" ? name.slice(3) : hyphenate(name.slice(2));  
 return [event, options];  
}  
var cachedNow = 0;  
var p = Promise.resolve();  
var getNow = () => cachedNow || (p.then(() => cachedNow = 0), cachedNow = Date.now());  
function createInvoker(initialValue, instance) {  
 const invoker = (e) => {  
 if (!e.\_vts) {  
 e.\_vts = Date.now();  
 } else if (e.\_vts <= invoker.attached) {  
 return;  
 }  
 callWithAsyncErrorHandling(  
 patchStopImmediatePropagation(e, invoker.value),  
 instance,  
 5,  
 [e]  
 );  
 };  
 invoker.value = initialValue;  
 invoker.attached = getNow();  
 return invoker;  
}  
function sanitizeEventValue(value, propName) {  
 if (isFunction(value) || isArray(value)) {  
 return value;  
 }  
 warn2(  
 `Wrong type passed as event handler to ${propName} - did you forget @ or : in front of your prop?  
Expected function or array of functions, received type ${typeof value}.`  
 );  
 return NOOP;  
}  
function patchStopImmediatePropagation(e, value) {  
 if (isArray(value)) {  
 const originalStop = e.stopImmediatePropagation;  
 e.stopImmediatePropagation = () => {  
 originalStop.call(e);  
 e.\_stopped = true;  
 };  
 return value.map(  
 (fn) => (e2) => !e2.\_stopped && fn && fn(e2)  
 );  
 } else {  
 return value;  
 }  
}  
var isNativeOn = (key) => key.charCodeAt(0) === 111 && key.charCodeAt(1) === 110 && // lowercase letter  
key.charCodeAt(2) > 96 && key.charCodeAt(2) < 123;  
var patchProp = (el, key, prevValue, nextValue, namespace, parentComponent) => {  
 const isSVG = namespace === "svg";  
 if (key === "class") {  
 patchClass(el, nextValue, isSVG);  
 } else if (key === "style") {  
 patchStyle(el, prevValue, nextValue);  
 } else if (isOn(key)) {  
 if (!isModelListener(key)) {  
 patchEvent(el, key, prevValue, nextValue, parentComponent);  
 }  
 } else if (key[0] === "." ? (key = key.slice(1), true) : key[0] === "^" ? (key = key.slice(1), false) : shouldSetAsProp(el, key, nextValue, isSVG)) {  
 patchDOMProp(el, key, nextValue);  
 if (!el.tagName.includes("-") && (key === "value" || key === "checked" || key === "selected")) {  
 patchAttr(el, key, nextValue, isSVG, parentComponent, key !== "value");  
 }  
 } else {  
 if (key === "true-value") {  
 el.\_trueValue = nextValue;  
 } else if (key === "false-value") {  
 el.\_falseValue = nextValue;  
 }  
 patchAttr(el, key, nextValue, isSVG);  
 }  
};  
function shouldSetAsProp(el, key, value, isSVG) {  
 if (isSVG) {  
 if (key === "innerHTML" || key === "textContent") {  
 return true;  
 }  
 if (key in el && isNativeOn(key) && isFunction(value)) {  
 return true;  
 }  
 return false;  
 }  
 if (key === "spellcheck" || key === "draggable" || key === "translate") {  
 return false;  
 }  
 if (key === "form") {  
 return false;  
 }  
 if (key === "list" && el.tagName === "INPUT") {  
 return false;  
 }  
 if (key === "type" && el.tagName === "TEXTAREA") {  
 return false;  
 }  
 if (key === "width" || key === "height") {  
 const tag = el.tagName;  
 if (tag === "IMG" || tag === "VIDEO" || tag === "CANVAS" || tag === "SOURCE") {  
 return false;  
 }  
 }  
 if (isNativeOn(key) && isString(value)) {  
 return false;  
 }  
 return key in el;  
}  
function defineCustomElement(options, extraOptions, hydrate2) {  
 const Comp = defineComponent(options, extraOptions);  
 class VueCustomElement extends VueElement {  
 constructor(initialProps) {  
 super(Comp, initialProps, hydrate2);  
 }  
 }  
 VueCustomElement.def = Comp;  
 return VueCustomElement;  
}  
var defineSSRCustomElement = (options, extraOptions) => {  
 return defineCustomElement(options, extraOptions, hydrate);  
};  
var BaseClass = typeof HTMLElement !== "undefined" ? HTMLElement : class {  
};  
var VueElement = class \_VueElement extends BaseClass {  
 constructor(\_def, \_props = {}, hydrate2) {  
 super();  
 this.\_def = \_def;  
 this.\_props = \_props;  
 this.\_instance = null;  
 this.\_connected = false;  
 this.\_resolved = false;  
 this.\_numberProps = null;  
 this.\_ob = null;  
 if (this.shadowRoot && hydrate2) {  
 hydrate2(this.\_createVNode(), this.shadowRoot);  
 } else {  
 if (this.shadowRoot) {  
 warn2(  
 `Custom element has pre-rendered declarative shadow root but is not defined as hydratable. Use \`defineSSRCustomElement\`.`  
 );  
 }  
 this.attachShadow({ mode: "open" });  
 if (!this.\_def.\_\_asyncLoader) {  
 this.\_resolveProps(this.\_def);  
 }  
 }  
 }  
 connectedCallback() {  
 this.\_connected = true;  
 if (!this.\_instance) {  
 if (this.\_resolved) {  
 this.\_update();  
 } else {  
 this.\_resolveDef();  
 }  
 }  
 }  
 disconnectedCallback() {  
 this.\_connected = false;  
 nextTick(() => {  
 if (!this.\_connected) {  
 if (this.\_ob) {  
 this.\_ob.disconnect();  
 this.\_ob = null;  
 }  
 render(null, this.shadowRoot);  
 this.\_instance = null;  
 }  
 });  
 }  
 \_resolveDef() {  
 this.\_resolved = true;  
 for (let i = 0; i < this.attributes.length; i++) {  
 this.\_setAttr(this.attributes[i].name);  
 }  
 this.\_ob = new MutationObserver((mutations) => {  
 for (const m of mutations) {  
 this.\_setAttr(m.attributeName);  
 }  
 });  
 this.\_ob.observe(this, { attributes: true });  
 const resolve2 = (def2, isAsync = false) => {  
 const { props, styles } = def2;  
 let numberProps;  
 if (props && !isArray(props)) {  
 for (const key in props) {  
 const opt = props[key];  
 if (opt === Number || opt && opt.type === Number) {  
 if (key in this.\_props) {  
 this.\_props[key] = toNumber(this.\_props[key]);  
 }  
 (numberProps || (numberProps = /\* @\_\_PURE\_\_ \*/ Object.create(null)))[camelize(key)] = true;  
 }  
 }  
 }  
 this.\_numberProps = numberProps;  
 if (isAsync) {  
 this.\_resolveProps(def2);  
 }  
 this.\_applyStyles(styles);  
 this.\_update();  
 };  
 const asyncDef = this.\_def.\_\_asyncLoader;  
 if (asyncDef) {  
 asyncDef().then((def2) => resolve2(def2, true));  
 } else {  
 resolve2(this.\_def);  
 }  
 }  
 \_resolveProps(def2) {  
 const { props } = def2;  
 const declaredPropKeys = isArray(props) ? props : Object.keys(props || {});  
 for (const key of Object.keys(this)) {  
 if (key[0] !== "\_" && declaredPropKeys.includes(key)) {  
 this.\_setProp(key, this[key], true, false);  
 }  
 }  
 for (const key of declaredPropKeys.map(camelize)) {  
 Object.defineProperty(this, key, {  
 get() {  
 return this.\_getProp(key);  
 },  
 set(val) {  
 this.\_setProp(key, val);  
 }  
 });  
 }  
 }  
 \_setAttr(key) {  
 let value = this.hasAttribute(key) ? this.getAttribute(key) : void 0;  
 const camelKey = camelize(key);  
 if (this.\_numberProps && this.\_numberProps[camelKey]) {  
 value = toNumber(value);  
 }  
 this.\_setProp(camelKey, value, false);  
 }  
 \_getProp(key) {  
 return this.\_props[key];  
 }  
 \_setProp(key, val, shouldReflect = true, shouldUpdate = true) {  
 if (val !== this.\_props[key]) {  
 this.\_props[key] = val;  
 if (shouldUpdate && this.\_instance) {  
 this.\_update();  
 }  
 if (shouldReflect) {  
 if (val === true) {  
 this.setAttribute(hyphenate(key), "");  
 } else if (typeof val === "string" || typeof val === "number") {  
 this.setAttribute(hyphenate(key), val + "");  
 } else if (!val) {  
 this.removeAttribute(hyphenate(key));  
 }  
 }  
 }  
 }  
 \_update() {  
 render(this.\_createVNode(), this.shadowRoot);  
 }  
 \_createVNode() {  
 const vnode = createVNode(this.\_def, extend({}, this.\_props));  
 if (!this.\_instance) {  
 vnode.ce = (instance) => {  
 this.\_instance = instance;  
 instance.isCE = true;  
 if (true) {  
 instance.ceReload = (newStyles) => {  
 if (this.\_styles) {  
 this.\_styles.forEach((s) => this.shadowRoot.removeChild(s));  
 this.\_styles.length = 0;  
 }  
 this.\_applyStyles(newStyles);  
 this.\_instance = null;  
 this.\_update();  
 };  
 }  
 const dispatch = (event, args) => {  
 this.dispatchEvent(  
 new CustomEvent(event, {  
 detail: args  
 })  
 );  
 };  
 instance.emit = (event, ...args) => {  
 dispatch(event, args);  
 if (hyphenate(event) !== event) {  
 dispatch(hyphenate(event), args);  
 }  
 };  
 let parent = this;  
 while (parent = parent && (parent.parentNode || parent.host)) {  
 if (parent instanceof \_VueElement) {  
 instance.parent = parent.\_instance;  
 instance.provides = parent.\_instance.provides;  
 break;  
 }  
 }  
 };  
 }  
 return vnode;  
 }  
 \_applyStyles(styles) {  
 if (styles) {  
 styles.forEach((css) => {  
 const s = document.createElement("style");  
 s.textContent = css;  
 this.shadowRoot.appendChild(s);  
 if (true) {  
 (this.\_styles || (this.\_styles = [])).push(s);  
 }  
 });  
 }  
 }  
};  
function useCssModule(name = "$style") {  
 {  
 const instance = getCurrentInstance();  
 if (!instance) {  
 warn2(`useCssModule must be called inside setup()`);  
 return EMPTY\_OBJ;  
 }  
 const modules = instance.type.\_\_cssModules;  
 if (!modules) {  
 warn2(`Current instance does not have CSS modules injected.`);  
 return EMPTY\_OBJ;  
 }  
 const mod = modules[name];  
 if (!mod) {  
 warn2(`Current instance does not have CSS module named "${name}".`);  
 return EMPTY\_OBJ;  
 }  
 return mod;  
 }  
}  
var positionMap = /\* @\_\_PURE\_\_ \*/ new WeakMap();  
var newPositionMap = /\* @\_\_PURE\_\_ \*/ new WeakMap();  
var moveCbKey = Symbol("\_moveCb");  
var enterCbKey2 = Symbol("\_enterCb");  
var TransitionGroupImpl = {  
 name: "TransitionGroup",  
 props: extend({}, TransitionPropsValidators, {  
 tag: String,  
 moveClass: String  
 }),  
 setup(props, { slots }) {  
 const instance = getCurrentInstance();  
 const state = useTransitionState();  
 let prevChildren;  
 let children;  
 onUpdated(() => {  
 if (!prevChildren.length) {  
 return;  
 }  
 const moveClass = props.moveClass || `${props.name || "v"}-move`;  
 if (!hasCSSTransform(  
 prevChildren[0].el,  
 instance.vnode.el,  
 moveClass  
 )) {  
 return;  
 }  
 prevChildren.forEach(callPendingCbs);  
 prevChildren.forEach(recordPosition);  
 const movedChildren = prevChildren.filter(applyTranslation);  
 forceReflow();  
 movedChildren.forEach((c) => {  
 const el = c.el;  
 const style = el.style;  
 addTransitionClass(el, moveClass);  
 style.transform = style.webkitTransform = style.transitionDuration = "";  
 const cb = el[moveCbKey] = (e) => {  
 if (e && e.target !== el) {  
 return;  
 }  
 if (!e || /transform$/.test(e.propertyName)) {  
 el.removeEventListener("transitionend", cb);  
 el[moveCbKey] = null;  
 removeTransitionClass(el, moveClass);  
 }  
 };  
 el.addEventListener("transitionend", cb);  
 });  
 });  
 return () => {  
 const rawProps = toRaw(props);  
 const cssTransitionProps = resolveTransitionProps(rawProps);  
 let tag = rawProps.tag || Fragment;  
 prevChildren = [];  
 if (children) {  
 for (let i = 0; i < children.length; i++) {  
 const child = children[i];  
 if (child.el && child.el instanceof Element) {  
 prevChildren.push(child);  
 setTransitionHooks(  
 child,  
 resolveTransitionHooks(  
 child,  
 cssTransitionProps,  
 state,  
 instance  
 )  
 );  
 positionMap.set(  
 child,  
 child.el.getBoundingClientRect()  
 );  
 }  
 }  
 }  
 children = slots.default ? getTransitionRawChildren(slots.default()) : [];  
 for (let i = 0; i < children.length; i++) {  
 const child = children[i];  
 if (child.key != null) {  
 setTransitionHooks(  
 child,  
 resolveTransitionHooks(child, cssTransitionProps, state, instance)  
 );  
 } else if (true) {  
 warn2(`<TransitionGroup> children must be keyed.`);  
 }  
 }  
 return createVNode(tag, null, children);  
 };  
 }  
};  
var removeMode = (props) => delete props.mode;  
removeMode(TransitionGroupImpl.props);  
var TransitionGroup = TransitionGroupImpl;  
function callPendingCbs(c) {  
 const el = c.el;  
 if (el[moveCbKey]) {  
 el[moveCbKey]();  
 }  
 if (el[enterCbKey2]) {  
 el[enterCbKey2]();  
 }  
}  
function recordPosition(c) {  
 newPositionMap.set(c, c.el.getBoundingClientRect());  
}  
function applyTranslation(c) {  
 const oldPos = positionMap.get(c);  
 const newPos = newPositionMap.get(c);  
 const dx = oldPos.left - newPos.left;  
 const dy = oldPos.top - newPos.top;  
 if (dx || dy) {  
 const s = c.el.style;  
 s.transform = s.webkitTransform = `translate(${dx}px,${dy}px)`;  
 s.transitionDuration = "0s";  
 return c;  
 }  
}  
function hasCSSTransform(el, root, moveClass) {  
 const clone = el.cloneNode();  
 const \_vtc = el[vtcKey];  
 if (\_vtc) {  
 \_vtc.forEach((cls) => {  
 cls.split(/\s+/).forEach((c) => c && clone.classList.remove(c));  
 });  
 }  
 moveClass.split(/\s+/).forEach((c) => c && clone.classList.add(c));  
 clone.style.display = "none";  
 const container = root.nodeType === 1 ? root : root.parentNode;  
 container.appendChild(clone);  
 const { hasTransform } = getTransitionInfo(clone);  
 container.removeChild(clone);  
 return hasTransform;  
}  
var getModelAssigner = (vnode) => {  
 const fn = vnode.props["onUpdate:modelValue"] || false;  
 return isArray(fn) ? (value) => invokeArrayFns(fn, value) : fn;  
};  
function onCompositionStart(e) {  
 e.target.composing = true;  
}  
function onCompositionEnd(e) {  
 const target = e.target;  
 if (target.composing) {  
 target.composing = false;  
 target.dispatchEvent(new Event("input"));  
 }  
}  
var assignKey = Symbol("\_assign");  
var vModelText = {  
 created(el, { modifiers: { lazy, trim, number } }, vnode) {  
 el[assignKey] = getModelAssigner(vnode);  
 const castToNumber = number || vnode.props && vnode.props.type === "number";  
 addEventListener(el, lazy ? "change" : "input", (e) => {  
 if (e.target.composing) return;  
 let domValue = el.value;  
 if (trim) {  
 domValue = domValue.trim();  
 }  
 if (castToNumber) {  
 domValue = looseToNumber(domValue);  
 }  
 el[assignKey](domValue);  
 });  
 if (trim) {  
 addEventListener(el, "change", () => {  
 el.value = el.value.trim();  
 });  
 }  
 if (!lazy) {  
 addEventListener(el, "compositionstart", onCompositionStart);  
 addEventListener(el, "compositionend", onCompositionEnd);  
 addEventListener(el, "change", onCompositionEnd);  
 }  
 },  
 mounted(el, { value }) {  
 el.value = value == null ? "" : value;  
 },  
 beforeUpdate(el, { value, oldValue, modifiers: { lazy, trim, number } }, vnode) {  
 el[assignKey] = getModelAssigner(vnode);  
 if (el.composing) return;  
 const elValue = (number || el.type === "number") && !/^0\d/.test(el.value) ? looseToNumber(el.value) : el.value;  
 const newValue = value == null ? "" : value;  
 if (elValue === newValue) {  
 return;  
 }  
 if (document.activeElement === el && el.type !== "range") {  
 if (lazy && value === oldValue) {  
 return;  
 }  
 if (trim && el.value.trim() === newValue) {  
 return;  
 }  
 }  
 el.value = newValue;  
 }  
};  
var vModelCheckbox = {  
 deep: true,  
 created(el, \_, vnode) {  
 el[assignKey] = getModelAssigner(vnode);  
 addEventListener(el, "change", () => {  
 const modelValue = el.\_modelValue;  
 const elementValue = getValue(el);  
 const checked = el.checked;  
 const assign = el[assignKey];  
 if (isArray(modelValue)) {  
 const index = looseIndexOf(modelValue, elementValue);  
 const found = index !== -1;  
 if (checked && !found) {  
 assign(modelValue.concat(elementValue));  
 } else if (!checked && found) {  
 const filtered = [...modelValue];  
 filtered.splice(index, 1);  
 assign(filtered);  
 }  
 } else if (isSet(modelValue)) {  
 const cloned = new Set(modelValue);  
 if (checked) {  
 cloned.add(elementValue);  
 } else {  
 cloned.delete(elementValue);  
 }  
 assign(cloned);  
 } else {  
 assign(getCheckboxValue(el, checked));  
 }  
 });  
 },  
 mounted: setChecked,  
 beforeUpdate(el, binding, vnode) {  
 el[assignKey] = getModelAssigner(vnode);  
 setChecked(el, binding, vnode);  
 }  
};  
function setChecked(el, { value, oldValue }, vnode) {  
 el.\_modelValue = value;  
 if (isArray(value)) {  
 el.checked = looseIndexOf(value, vnode.props.value) > -1;  
 } else if (isSet(value)) {  
 el.checked = value.has(vnode.props.value);  
 } else if (value !== oldValue) {  
 el.checked = looseEqual(value, getCheckboxValue(el, true));  
 }  
}  
var vModelRadio = {  
 created(el, { value }, vnode) {  
 el.checked = looseEqual(value, vnode.props.value);  
 el[assignKey] = getModelAssigner(vnode);  
 addEventListener(el, "change", () => {  
 el[assignKey](getValue(el));  
 });  
 },  
 beforeUpdate(el, { value, oldValue }, vnode) {  
 el[assignKey] = getModelAssigner(vnode);  
 if (value !== oldValue) {  
 el.checked = looseEqual(value, vnode.props.value);  
 }  
 }  
};  
var vModelSelect = {  
 deep: true,  
 created(el, { value, modifiers: { number } }, vnode) {  
 const isSetModel = isSet(value);  
 addEventListener(el, "change", () => {  
 const selectedVal = Array.prototype.filter.call(el.options, (o) => o.selected).map(  
 (o) => number ? looseToNumber(getValue(o)) : getValue(o)  
 );  
 el[assignKey](  
 el.multiple ? isSetModel ? new Set(selectedVal) : selectedVal : selectedVal[0]  
 );  
 el.\_assigning = true;  
 nextTick(() => {  
 el.\_assigning = false;  
 });  
 });  
 el[assignKey] = getModelAssigner(vnode);  
 },  
 mounted(el, { value, modifiers: { number } }) {  
 setSelected(el, value);  
 },  
 beforeUpdate(el, \_binding, vnode) {  
 el[assignKey] = getModelAssigner(vnode);  
 },  
 updated(el, { value, modifiers: { number } }) {  
 if (!el.\_assigning) {  
 setSelected(el, value);  
 }  
 }  
};  
function setSelected(el, value, number) {  
 const isMultiple = el.multiple;  
 const isArrayValue = isArray(value);  
 if (isMultiple && !isArrayValue && !isSet(value)) {  
 warn2(  
 `<select multiple v-model> expects an Array or Set value for its binding, but got ${Object.prototype.toString.call(value).slice(8, -1)}.`  
 );  
 return;  
 }  
 for (let i = 0, l = el.options.length; i < l; i++) {  
 const option = el.options[i];  
 const optionValue = getValue(option);  
 if (isMultiple) {  
 if (isArrayValue) {  
 const optionType = typeof optionValue;  
 if (optionType === "string" || optionType === "number") {  
 option.selected = value.some((v) => String(v) === String(optionValue));  
 } else {  
 option.selected = looseIndexOf(value, optionValue) > -1;  
 }  
 } else {  
 option.selected = value.has(optionValue);  
 }  
 } else if (looseEqual(getValue(option), value)) {  
 if (el.selectedIndex !== i) el.selectedIndex = i;  
 return;  
 }  
 }  
 if (!isMultiple && el.selectedIndex !== -1) {  
 el.selectedIndex = -1;  
 }  
}  
function getValue(el) {  
 return "\_value" in el ? el.\_value : el.value;  
}  
function getCheckboxValue(el, checked) {  
 const key = checked ? "\_trueValue" : "\_falseValue";  
 return key in el ? el[key] : checked;  
}  
var vModelDynamic = {  
 created(el, binding, vnode) {  
 callModelHook(el, binding, vnode, null, "created");  
 },  
 mounted(el, binding, vnode) {  
 callModelHook(el, binding, vnode, null, "mounted");  
 },  
 beforeUpdate(el, binding, vnode, prevVNode) {  
 callModelHook(el, binding, vnode, prevVNode, "beforeUpdate");  
 },  
 updated(el, binding, vnode, prevVNode) {  
 callModelHook(el, binding, vnode, prevVNode, "updated");  
 }  
};  
function resolveDynamicModel(tagName, type) {  
 switch (tagName) {  
 case "SELECT":  
 return vModelSelect;  
 case "TEXTAREA":  
 return vModelText;  
 default:  
 switch (type) {  
 case "checkbox":  
 return vModelCheckbox;  
 case "radio":  
 return vModelRadio;  
 default:  
 return vModelText;  
 }  
 }  
}  
function callModelHook(el, binding, vnode, prevVNode, hook) {  
 const modelToUse = resolveDynamicModel(  
 el.tagName,  
 vnode.props && vnode.props.type  
 );  
 const fn = modelToUse[hook];  
 fn && fn(el, binding, vnode, prevVNode);  
}  
function initVModelForSSR() {  
 vModelText.getSSRProps = ({ value }) => ({ value });  
 vModelRadio.getSSRProps = ({ value }, vnode) => {  
 if (vnode.props && looseEqual(vnode.props.value, value)) {  
 return { checked: true };  
 }  
 };  
 vModelCheckbox.getSSRProps = ({ value }, vnode) => {  
 if (isArray(value)) {  
 if (vnode.props && looseIndexOf(value, vnode.props.value) > -1) {  
 return { checked: true };  
 }  
 } else if (isSet(value)) {  
 if (vnode.props && value.has(vnode.props.value)) {  
 return { checked: true };  
 }  
 } else if (value) {  
 return { checked: true };  
 }  
 };  
 vModelDynamic.getSSRProps = (binding, vnode) => {  
 if (typeof vnode.type !== "string") {  
 return;  
 }  
 const modelToUse = resolveDynamicModel(  
 vnode.type.toUpperCase(),  
 vnode.props && vnode.props.type  
 );  
 if (modelToUse.getSSRProps) {  
 return modelToUse.getSSRProps(binding, vnode);  
 }  
 };  
}  
var systemModifiers = ["ctrl", "shift", "alt", "meta"];  
var modifierGuards = {  
 stop: (e) => e.stopPropagation(),  
 prevent: (e) => e.preventDefault(),  
 self: (e) => e.target !== e.currentTarget,  
 ctrl: (e) => !e.ctrlKey,  
 shift: (e) => !e.shiftKey,  
 alt: (e) => !e.altKey,  
 meta: (e) => !e.metaKey,  
 left: (e) => "button" in e && e.button !== 0,  
 middle: (e) => "button" in e && e.button !== 1,  
 right: (e) => "button" in e && e.button !== 2,  
 exact: (e, modifiers) => systemModifiers.some((m) => e[`${m}Key`] && !modifiers.includes(m))  
};  
var withModifiers = (fn, modifiers) => {  
 const cache = fn.\_withMods || (fn.\_withMods = {});  
 const cacheKey = modifiers.join(".");  
 return cache[cacheKey] || (cache[cacheKey] = (event, ...args) => {  
 for (let i = 0; i < modifiers.length; i++) {  
 const guard = modifierGuards[modifiers[i]];  
 if (guard && guard(event, modifiers)) return;  
 }  
 return fn(event, ...args);  
 });  
};  
var keyNames = {  
 esc: "escape",  
 space: " ",  
 up: "arrow-up",  
 left: "arrow-left",  
 right: "arrow-right",  
 down: "arrow-down",  
 delete: "backspace"  
};  
var withKeys = (fn, modifiers) => {  
 const cache = fn.\_withKeys || (fn.\_withKeys = {});  
 const cacheKey = modifiers.join(".");  
 return cache[cacheKey] || (cache[cacheKey] = (event) => {  
 if (!("key" in event)) {  
 return;  
 }  
 const eventKey = hyphenate(event.key);  
 if (modifiers.some((k) => k === eventKey || keyNames[k] === eventKey)) {  
 return fn(event);  
 }  
 });  
};  
var rendererOptions = extend({ patchProp }, nodeOps);  
var renderer;  
var enabledHydration = false;  
function ensureRenderer() {  
 return renderer || (renderer = createRenderer(rendererOptions));  
}  
function ensureHydrationRenderer() {  
 renderer = enabledHydration ? renderer : createHydrationRenderer(rendererOptions);  
 enabledHydration = true;  
 return renderer;  
}  
var render = (...args) => {  
 ensureRenderer().render(...args);  
};  
var hydrate = (...args) => {  
 ensureHydrationRenderer().hydrate(...args);  
};  
var createApp = (...args) => {  
 const app = ensureRenderer().createApp(...args);  
 if (true) {  
 injectNativeTagCheck(app);  
 injectCompilerOptionsCheck(app);  
 }  
 const { mount } = app;  
 app.mount = (containerOrSelector) => {  
 const container = normalizeContainer(containerOrSelector);  
 if (!container) return;  
 const component = app.\_component;  
 if (!isFunction(component) && !component.render && !component.template) {  
 component.template = container.innerHTML;  
 }  
 container.innerHTML = "";  
 const proxy = mount(container, false, resolveRootNamespace(container));  
 if (container instanceof Element) {  
 container.removeAttribute("v-cloak");  
 container.setAttribute("data-v-app", "");  
 }  
 return proxy;  
 };  
 return app;  
};  
var createSSRApp = (...args) => {  
 const app = ensureHydrationRenderer().createApp(...args);  
 if (true) {  
 injectNativeTagCheck(app);  
 injectCompilerOptionsCheck(app);  
 }  
 const { mount } = app;  
 app.mount = (containerOrSelector) => {  
 const container = normalizeContainer(containerOrSelector);  
 if (container) {  
 return mount(container, true, resolveRootNamespace(container));  
 }  
 };  
 return app;  
};  
function resolveRootNamespace(container) {  
 if (container instanceof SVGElement) {  
 return "svg";  
 }  
 if (typeof MathMLElement === "function" && container instanceof MathMLElement) {  
 return "mathml";  
 }  
}  
function injectNativeTagCheck(app) {  
 Object.defineProperty(app.config, "isNativeTag", {  
 value: (tag) => isHTMLTag(tag) || isSVGTag(tag) || isMathMLTag(tag),  
 writable: false  
 });  
}  
function injectCompilerOptionsCheck(app) {  
 if (isRuntimeOnly()) {  
 const isCustomElement = app.config.isCustomElement;  
 Object.defineProperty(app.config, "isCustomElement", {  
 get() {  
 return isCustomElement;  
 },  
 set() {  
 warn2(  
 `The \`isCustomElement\` config option is deprecated. Use \`compilerOptions.isCustomElement\` instead.`  
 );  
 }  
 });  
 const compilerOptions = app.config.compilerOptions;  
 const msg = `The \`compilerOptions\` config option is only respected when using a build of Vue.js that includes the runtime compiler (aka "full build"). Since you are using the runtime-only build, \`compilerOptions\` must be passed to \`@vue/compiler-dom\` in the build setup instead.  
- For vue-loader: pass it via vue-loader's \`compilerOptions\` loader option.  
- For vue-cli: see https://cli.vuejs.org/guide/webpack.html#modifying-options-of-a-loader  
- For vite: pass it via @vitejs/plugin-vue options. See https://github.com/vitejs/vite-plugin-vue/tree/main/packages/plugin-vue#example-for-passing-options-to-vuecompiler-sfc`;  
 Object.defineProperty(app.config, "compilerOptions", {  
 get() {  
 warn2(msg);  
 return compilerOptions;  
 },  
 set() {  
 warn2(msg);  
 }  
 });  
 }  
}  
function normalizeContainer(container) {  
 if (isString(container)) {  
 const res = document.querySelector(container);  
 if (!res) {  
 warn2(  
 `Failed to mount app: mount target selector "${container}" returned null.`  
 );  
 }  
 return res;  
 }  
 if (window.ShadowRoot && container instanceof window.ShadowRoot && container.mode === "closed") {  
 warn2(  
 `mounting on a ShadowRoot with \`{mode: "closed"}\` may lead to unpredictable bugs`  
 );  
 }  
 return container;  
}  
var ssrDirectiveInitialized = false;  
var initDirectivesForSSR = () => {  
 if (!ssrDirectiveInitialized) {  
 ssrDirectiveInitialized = true;  
 initVModelForSSR();  
 initVShowForSSR();  
 }  
};  
function initDev() {  
 {  
 initCustomFormatter();  
 }  
}  
if (true) {  
 initDev();  
}  
var compile2 = () => {  
 if (true) {  
 warn2(  
 `Runtime compilation is not supported in this build of Vue. Configure your bundler to alias "vue" to "vue/dist/vue.esm-bundler.js".`  
 );  
 }  
};  
export {  
 camelize,  
 capitalize,  
 toHandlerKey,  
 normalizeStyle,  
 normalizeClass,  
 normalizeProps,  
 toDisplayString,  
 EffectScope,  
 effectScope,  
 getCurrentScope,  
 onScopeDispose,  
 ReactiveEffect,  
 effect,  
 stop,  
 reactive,  
 shallowReactive,  
 readonly,  
 shallowReadonly,  
 isReactive,  
 isReadonly,  
 isShallow,  
 isProxy,  
 toRaw,  
 markRaw,  
 isRef2 as isRef,  
 ref,  
 shallowRef,  
 triggerRef,  
 unref,  
 toValue,  
 proxyRefs,  
 customRef,  
 toRefs,  
 toRef,  
 TrackOpTypes,  
 TriggerOpTypes,  
 assertNumber,  
 ErrorCodes,  
 callWithErrorHandling,  
 callWithAsyncErrorHandling,  
 handleError,  
 nextTick,  
 queuePostFlushCb,  
 pushScopeId,  
 popScopeId,  
 withScopeId,  
 withCtx,  
 withDirectives,  
 useTransitionState,  
 BaseTransitionPropsValidators,  
 BaseTransition,  
 resolveTransitionHooks,  
 setTransitionHooks,  
 getTransitionRawChildren,  
 defineComponent,  
 defineAsyncComponent,  
 KeepAlive,  
 onActivated,  
 onDeactivated,  
 onBeforeMount,  
 onMounted,  
 onBeforeUpdate,  
 onUpdated,  
 onBeforeUnmount,  
 onUnmounted,  
 onServerPrefetch,  
 onRenderTriggered,  
 onRenderTracked,  
 onErrorCaptured,  
 resolveComponent,  
 resolveDynamicComponent,  
 resolveDirective,  
 renderList,  
 createSlots,  
 renderSlot,  
 toHandlers,  
 defineProps,  
 defineEmits,  
 defineExpose,  
 defineOptions,  
 defineSlots,  
 defineModel,  
 withDefaults,  
 useSlots,  
 useAttrs,  
 mergeDefaults,  
 mergeModels,  
 createPropsRestProxy,  
 withAsyncContext,  
 provide,  
 inject,  
 hasInjectionContext,  
 Teleport,  
 createRenderer,  
 createHydrationRenderer,  
 ssrContextKey,  
 useSSRContext,  
 watchEffect,  
 watchPostEffect,  
 watchSyncEffect,  
 watch,  
 useModel,  
 Suspense,  
 Fragment,  
 Text,  
 Comment,  
 Static,  
 openBlock,  
 setBlockTracking,  
 createElementBlock,  
 createBlock,  
 isVNode,  
 transformVNodeArgs,  
 createBaseVNode,  
 createVNode,  
 guardReactiveProps,  
 cloneVNode,  
 createTextVNode,  
 createStaticVNode,  
 createCommentVNode,  
 mergeProps,  
 getCurrentInstance,  
 registerRuntimeCompiler,  
 isRuntimeOnly,  
 computed2 as computed,  
 h,  
 initCustomFormatter,  
 withMemo,  
 isMemoSame,  
 version,  
 warn2 as warn,  
 ErrorTypeStrings,  
 devtools,  
 setDevtoolsHook,  
 ssrUtils,  
 resolveFilter,  
 compatUtils,  
 DeprecationTypes,  
 Transition,  
 vShow,  
 useCssVars,  
 defineCustomElement,  
 defineSSRCustomElement,  
 VueElement,  
 useCssModule,  
 TransitionGroup,  
 vModelText,  
 vModelCheckbox,  
 vModelRadio,  
 vModelSelect,  
 vModelDynamic,  
 withModifiers,  
 withKeys,  
 render,  
 hydrate,  
 createApp,  
 createSSRApp,  
 initDirectivesForSSR,  
 compile2 as compile  
};  
@vue/shared/dist/shared.esm-bundler.js:  
 (\*\*  
 (\*! #\_\_NO\_SIDE\_EFFECTS\_\_ \*)  
@vue/reactivity/dist/reactivity.esm-bundler.js:  
 (\*\*  
@vue/runtime-core/dist/runtime-core.esm-bundler.js:  
 (\*\*  
 (\*! #\_\_NO\_SIDE\_EFFECTS\_\_ \*)  
@vue/runtime-dom/dist/runtime-dom.esm-bundler.js:  
 (\*\*  
 (\*! #\_\_NO\_SIDE\_EFFECTS\_\_ \*)  
vue/dist/vue.runtime.esm-bundler.js:  
 (\*\*