

Vue.js3対応



# Vue3の特徴

# Vue3の特徴



メイン・・・大規模開発の対応

1. 処理の高速化
2. ファイルサイズ減
3. Provide/Inject・・・長距離のprops
4. 大規模対応・・・CompositionAPI
5. TypeScriptサポート改善
6. IE11未対応

# Vue3 ホームページ



Vue3 ホームページ(ガイド)

<https://v3.ja.vuejs.org/>

Vue3 API

<https://v3.ja.vuejs.org/api/>

# Vue2とVue3の起動の違い

---

Vue2

```
new Vue({  
  el: '#app'  
})
```

Vue3

```
Vue.createApp({  
}).mount('#app')
```



# Vue3 インスタンス化 CDN

---

```
<div id="counter">  
  {{ test }}  
</div>
```

```
<script src="https://unpkg.com/vue@next"></script>  
<script>  
  Vue.createApp({  
    data(){  
      return{  
        counter : 0  
      }  
    }  
  }).mount('#counter');  
</script>
```

# Vue3 対応バージョン

---

GoogleChromeの Vue.js devtools

Vue2 ・ ・ ver5.x Vue3 ・ ・ ver6.x (beta)

VueCLI ・ ・ ver4.x

VueRouter ・ ・ ver4.x

Vuex ・ ・ ver4.x

# VueCLI



バージョン確認

```
$ vue --version
```

バージョンアップ(VueCLI ver3以降)

```
$ npm install -g @vue/cli
```


プロジェクト作成

```
$ vue create vue3-test
```

(manualでvue3, babel, router, vuex, lint にチェック)



# エントリーポイントの確認



src/main.js

```
import { createApp } from 'vue' //必要な機能を指定
import App from './App.vue'
import router from './router'
import store from './store'
```

```
createApp(App).use(store).use(router).mount('#app')
```

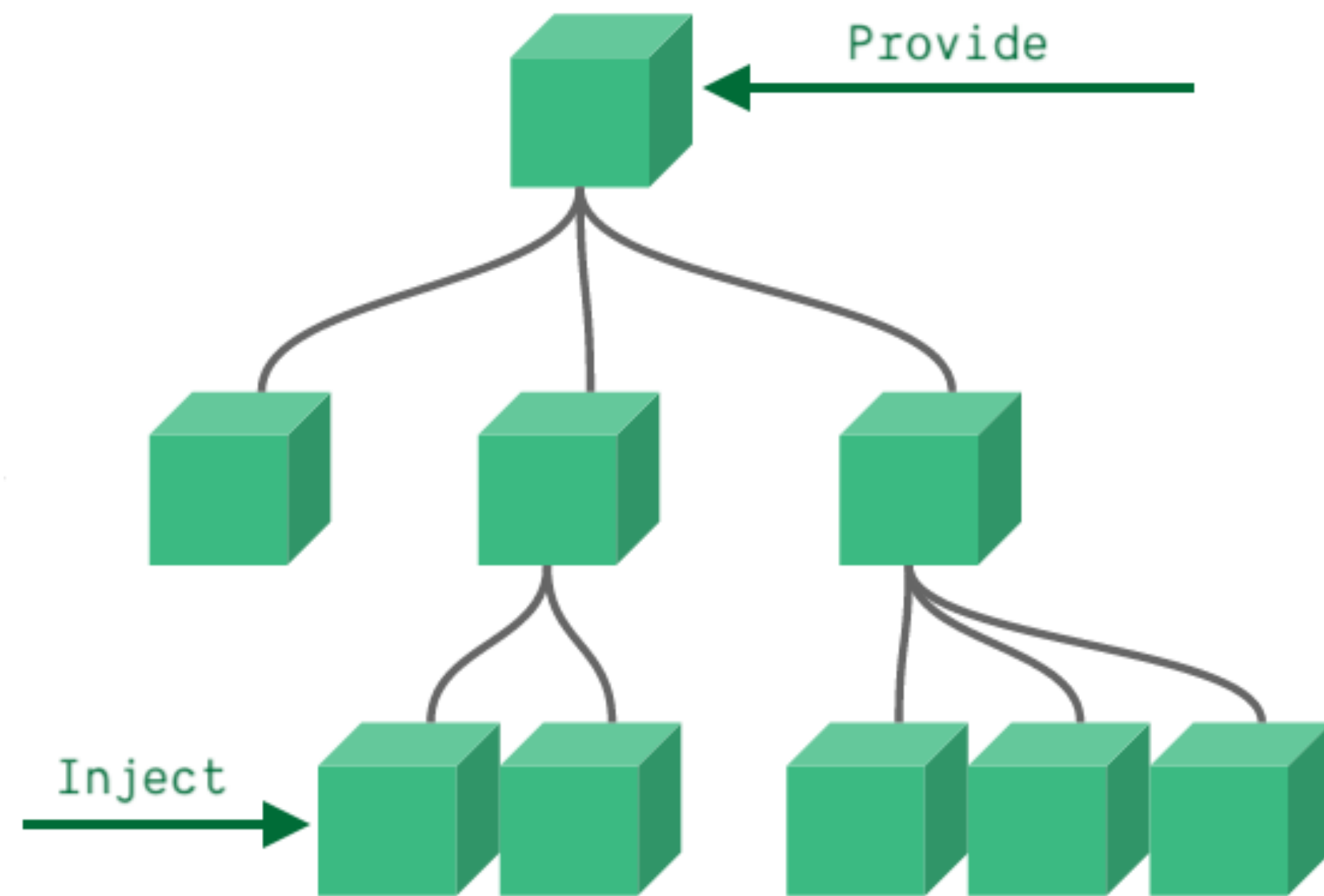
createAppは  
アプリケーションAPIをチェーンで繋げる事ができる



Provide/Inject

# Provide(提供)/Inject(注入)

親->孫へデータを渡せる  
長距離props



親 App.vue  

```
provide(){  
  return {  
    userName: '親で設定した値'  
  }  
}
```

子 Children.vue  
孫 GrandChildren.vue  

```
inject: ['userName']
```

# templateのタグ仕様

## Vue2

1つdivを書き、その中にコードを書いていく

```
<template>  
  <div>  
    ～略～  
  </div>  
</template>
```

## Vue3

Template直下に複数のタグを書ける

```
<tempalte>  
  <div></div>  
  <div></div>  
  <div></div>  
</temaplate>
```

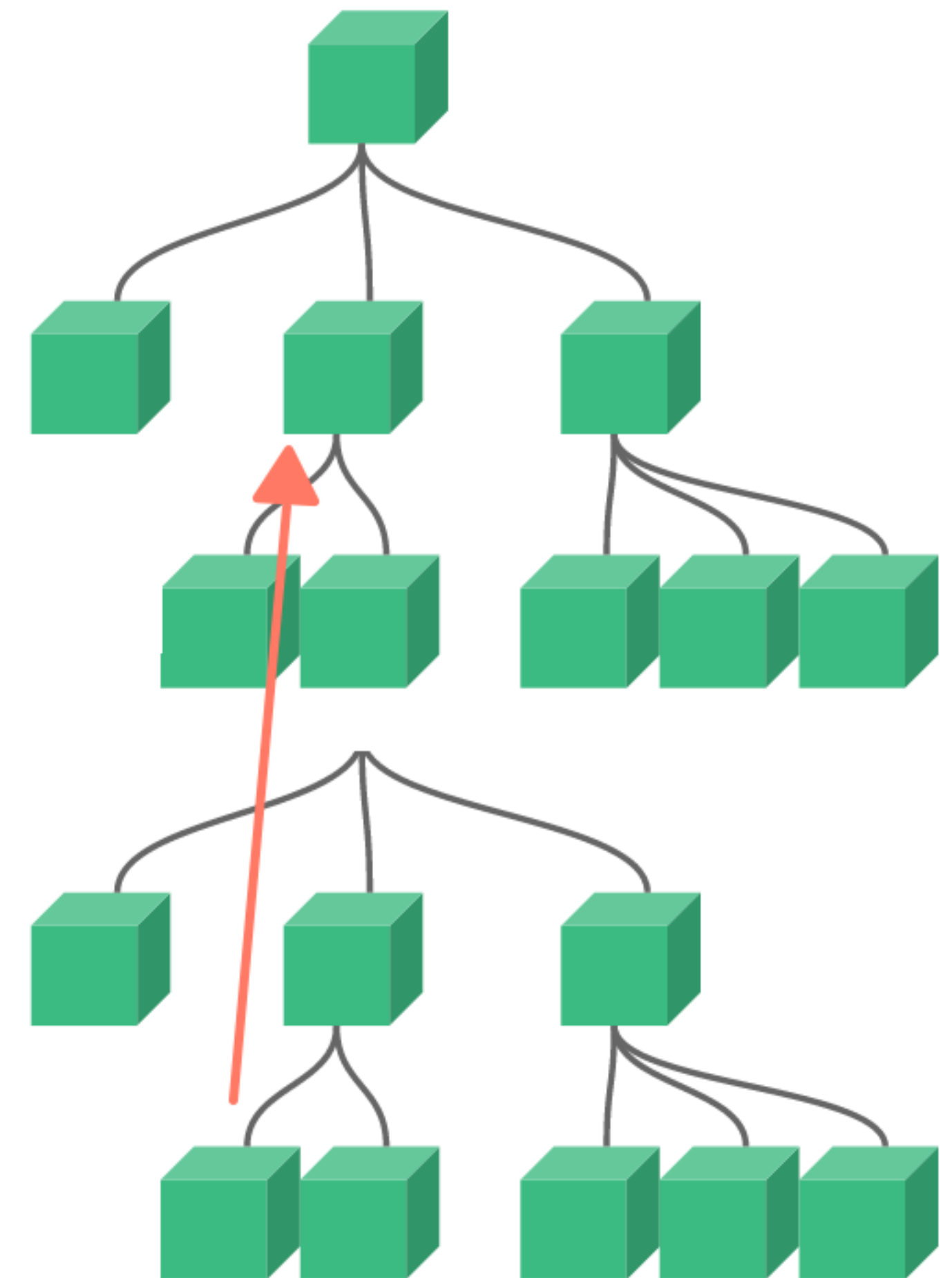


# Teleport

# Teleport

親子関係を飛び越えて  
表示できる機能

使い方・・・  
モーダルウィンドウ





# Teleport (template部)

```
<template>
  <div class="relative">
    <button @click="modalOpen = true">モーダル</button>
    <teleport to="body">
      <div v-if="modalOpen" class="modal">
        <div>
          <p>モーダルウィンドウ</p>
          <button @click="modalOpen = false">
            閉じる
          </button>
        </div>
      </div>
    </teleport>
  </div>
</template>
```

# Teleport (JS・CSS部)

```
<script>
export default {
  data(){
    return{
      modalOpen: false
    }
  }
}
</script>
```

```
<style scoped>
.relative{
  position:relative;
}
```

```
.modal {
  position: absolute;
  top: 0; right: 0; bottom: 0; left: 0;
  background-color: rgba(0,0,0,.5);
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
}
```

```
.modal div {
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  background-color: white;
  width: 300px;
  height: 300px;
  padding: 5px;
}
</style>
```



# CompositionAPI

# CompositionAPI



Composition(構成、合成)

1. 大規模対応
2. コードの再利用性 (合成関数)
3. TypeScriptのサポート

# API 比較イメージ図

## Options API

```
import { createApp } from 'vue'
import { createRouter } from 'vue-router'
import { createPinia } from 'pinia'

import App from './App.vue'
import routes from './routes'

const app = createApp(App)
const router = createRouter({
  routes,
})
const pinia = createPinia()

app.use(router)
app.use(pinia)

app.mount('#app')
```

## Composition API

```
import { createApp } from 'vue'
import { createRouter } from 'vue-router'
import { createPinia } from 'pinia'

import App from './App.vue'
import routes from './routes'

const app = createApp(App)
const router = createRouter({
  routes,
})
const pinia = createPinia()

app.use(router)
app.use(pinia)

app.mount('#app')
```

OptionsAPI  
data, methods,  
computed, mounted  
など、役割ごとに  
場所がわかれていた

CompositionAPI  
処理ごとにまとめて  
書くことができる

# OptionsAPIに含まれる要素

---

- data
- computed / watch
- methods
- lifecycle methods
- props / emit

これまでのOptionsAPIは引き続き使用可能  
CompositionAPIとも併用可能



# setup関数の実行タイミング

```
export default {  
  setup(){  
    console.log('setup') // createdより早い  
  },  
  created(){  
    console.log('created')  
  },  
  mounted(){  
    console.log('mounted')  
  }  
}
```

# Setup関数内のthisは不可

```
export default{  
  setup(){  
    console.log(this)  
  }  
</script>
```

コンソール上に undefinedと表示  
Setup関数内はthisを使わない  
->アロー関数がいやすい

# Setup関数の戻り値

```
<template>
  {{ name }}
</template>
```

```
<script>
export default{
  setup(){
    let name = '大谷'

    return { name } // returnに書いた変数・関数をtemplate内で扱える
  }
}
</script>
```



# リアクティブ

# リアクティブ 比較表

	ref	reactive	reactive return ...toRefs()
対象	プリミティブな値 (文字、数値など)	オブジェクト (dataに近い)	同左 (合成関数でよく使われる)
設定	<code>const nameRef = ref('錦織')</code> ※オブジェクトでラップ	<code>const book = reactive({   title: 'タイトル',   author: ['大谷', '伊藤'] })</code>	同左
template内で指定	<code>{{ nameRef }}</code>	<code>{{ book.title }}</code>	<code>{{ title }}</code>
Script内で扱う場合	<code>nameRef.value</code>	<code>book.title</code>	<code>title</code>
return時	<code>return { nameRef }</code>	<code>return { book }</code>	<code>return { ...   toRefs(book) }</code>

# ref リアクティブな変数

1つの値をリアクティブにしたい場合に使う  
値を参照オブジェクトでラップすることで  
リアクティブな状態にする

```
const nameRef = ref('錦織')
```

```
console.log(nameRef) // 参照オブジェクト RefImpl  
console.log(nameRef.value) // 値 RefImpl.value
```

値を取得するには `.value` が必要



# reactive リアクティブなオブジェクト



dataに近い。オブジェクトで指定する

```
const book = reactive({  
  title: 'タイトル',  
  author: ['大谷', '伊藤'],  
})
```

こちらはvalue不要  
template内, script内 ともに  
book.title などで表示できる

# toRefs



reactiveオブジェクトを展開して使う場合

```
return { ...book }
```

とするとリアクティブではなくなる

リアクティブを維持したまま展開するには  


```
return { ...toRefs(book) }
```

`book.title` ではなく、`title` として表示できる



# メソッド～ ライフサイクル

# メソッド その1



JSのまま書くことができる


JSは変数に関数を含めることができるので

```
<button @click="btnClicked">クリック</button>
```

```
const btnClicked = () => {  
  console.log('クリック')  
}
```

```
return { btnClicked }
```

# メソッド その2



Setup関数内の変数もそのまま指定可能(this不要)

```
const btnClicked = () => {  
  console.log(book.title)  
}
```

引数にeでイベント取得

```
const btnClicked = e => {  
  console.log(e)  
}
```

# Computed



```
import { reactive, computed } from 'vue'
```

```
const item = reactive({  
  price: 100,  
  number: 1  
})
```

```
const totalPrice = computed(()=>{  
  return item.price * item.number //必ずreturnが必要  
})
```

```
return {  
  item, totalPrice  
}
```



# Computed (get/set)

Vue3 API reactivityAPI -> computed and watch より

<https://v3.vuejs.org/api/computed-watch-api.html#computed>

```
const count = ref(1)
const plusOne = computed({
  get: () => count.value + 1,
  set: val => {
    count.value = val - 1
  }
})
```

```
plusOne.value = 1
console.log(count.value) // 0
```

# watch 比較表

	watch	watchEffect
監視対象	第1引数 (引数内に配列で複数指定可能) watch(price, ()=>{ })	関数内の リアクティブオブジェクト
取得できる値	第2引数の()内に (newValue, oldValue) などと指定	なし
初回実行のタイミング	監視対象オブジェクトが 変更されたタイミング (Lazy Load)	定義時(画面更新時)に実行
使い勝手	OptionsAPIと同様 少し複雑	シンプル

# watch



```
<div>watch: <input v-model="search">{{ search }}</div>
```

```
<script>
```

```
import { watch } from 'vue'
```

```
const search = ref("")
```

```
  watch(search, (newValue, prevValue)=>{
```

```
    console.log(`watch: ${search.value}`)
```

```
    console.log(`new: ${newValue}`)
```

```
    console.log(`prev: ${prevValue}`)
```

```
  })
```

```
return { search }
```

# watchEffect



```
<div>watchEffect:  
<input v-model="searchEffect">{{ searchEffect }}</div>
```

```
<script>  
import { watchEffect } from 'vue'  
  
const searchEffect = ref("")  
  watchEffect(=>{  
    console.log(`watchEffect: ${searchEffect.value}`)  
  })  
  
return { searchEffect }
```

# setup内のライフサイクルフック

オプション API	setup 内のフック
beforeCreate	不要*
created	不要*
beforeMount	onBeforeMount
mounted	onMounted
beforeUpdate	onBeforeUpdate
updated	onUpdated
beforeUnmount	onBeforeUnmount
unmounted	onUnmounted
errorCaptured	onErrorCaptured
renderTracked	onRenderTracked
renderTriggered	onRenderTriggered
activated	onActivated
deactivated	onDeactivated

setupは  
beforeCreateとcreatedの  
ライフサイクルで実行される

beforeCreate  
created  
は書く必要がない

# ライフサイクルフック

---

OptionsAPIと同じ名前だけれど  
頭にonがつく

```
import { onMounted } from 'vue'
```

```
onMounted(() => {  
  console.log('onMounted')  
})
```





# setup関数の 引数

# setupの第一引数 props

---

```
props:{  
  books: Array  
},  
setup(props, context){  
  console.log(props.books)  
}
```

Props ・ ・ リアクティブ。分割代入はNG。  
設定したいならtoRef() またはtoRefs()

# setupの引数 context

thisは使わず、代わりにcontextオブジェクトで指定する

```
setup(props, context){  
  console.log(context.emit)  
  console.log(context.attr)  
  console.log(context.slots)  
}
```

contextはJSオブジェクト 分割代入可能

```
setup(props, { attr, slots, emit }){  
  console.log(emit())  
}
```

Props不要な場合は\_と記載

```
setup(_, { emit }){
```

```
}
```

# setupの引数 context.emit

```
// 子コンポーネント
setup(props, { emit }){
  const emitTest = () => {
    emit('custom-event', '子の値')
  }
}
```

```
  return { emitTest }
}
```

```
//親コンポーネント
<PropsEmitTest
  @custom-event="parentMethod"
/>
```



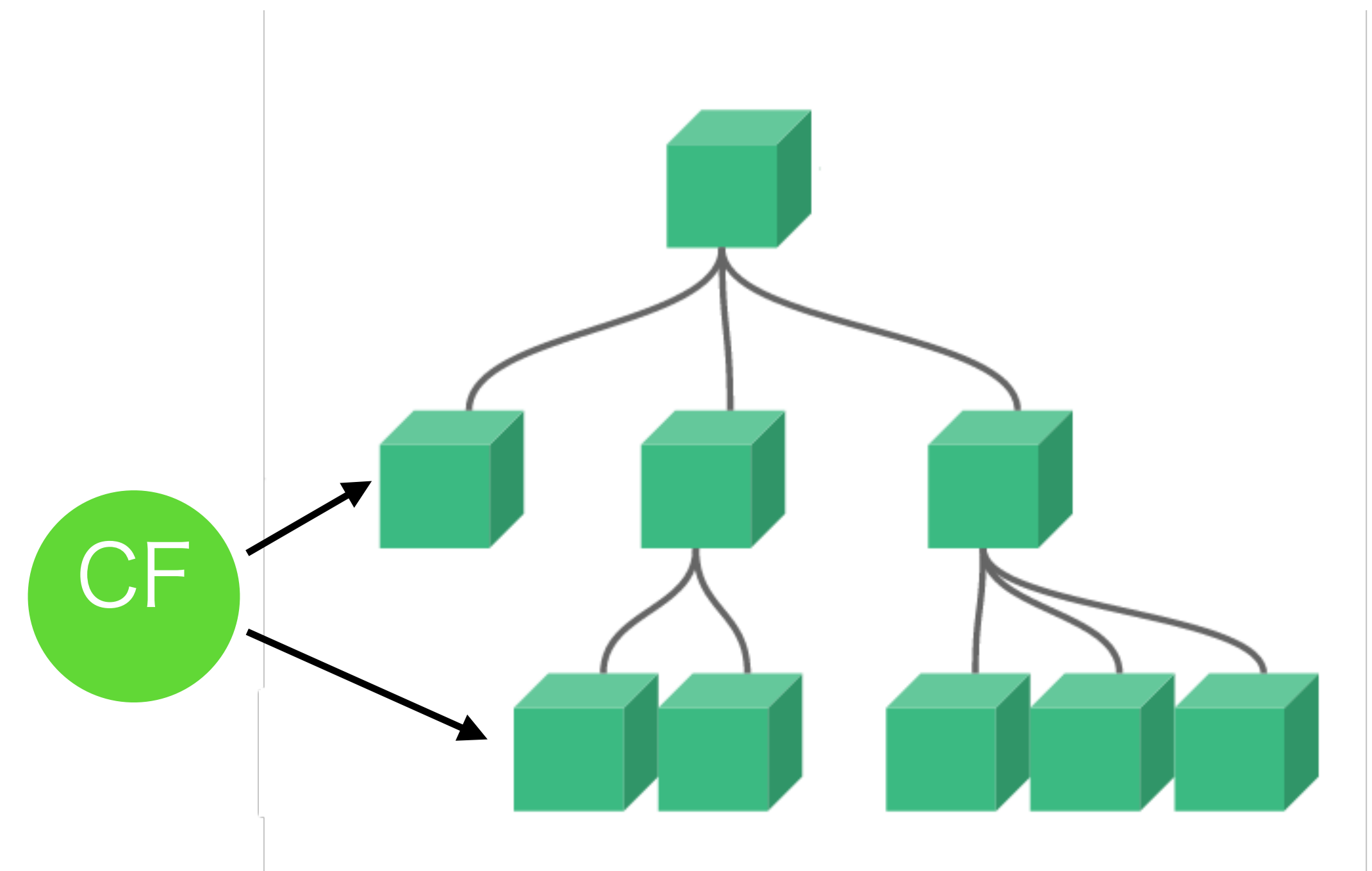
# Composition Function (合成関数)

# 合成関数 Composition Function

compositionAPIを含む関数

関数として作成し  
Setupの中で読み込む

モジュール化すると  
他コンポーネントでも使える





# Composition Function

```
const useCounter = item => { //関数名の頭にuseをつける
  const increment = () => {
    item.amount++
  }
  const decrement = () => {
    item.amount—
  }
  const totalPrice = computed(()=>{
    return item.price * item.amount
  })

  // return で作成した変数(関数)を返す
  return { increment, decrement, totalPrice }
}
```

# Setup内で読み込む(script内)

```
import { reactive } from 'vue'
```

～useCounter 略～

```
setup(){  
  const item = reactive({  
    name: '商品名',  
    price: 100,  
    amount: 0  
  })  
  //分割代入  
  const { increment, decrement, totalPrice } = useCounter(item)  
  
  return { item, increment, decrement, totalPrice }  
}
```

# template側

```
<template>
  <div>
    <p>商品名:{{ item.name }}</p>
    <p>単価:{{ item.price }}</p>
    <p>合計: {{ totalPrice }}</p>
    <div>数量</div>
    <button @click='decrement'>-</button>
    <button @click='increment'>+</button>
  </div>
</template>
```

# モジュール化

src/composables フォルダ作成し useCounter.js を作成

```
import { computed } from 'vue'
export default function useCounter(item){
  const increment = () => {
    item.amount++
  }

  const decrement = () => {
    item.amount--
  }

  const totalPrice = computed(()=>{
    return item.price * item.amount
  })

  return {
    increment,
    decrement,
    totalPrice
  }
}
```

# モジュールを読み込む

```
import useCounter from '@composables/useCounter'
export default {
  setup() {
    const item = 略
    const { increment, decrement, totalPrice } = useCounter(item)

    return { item, increment, decrement, totalPrice }
  }
}
```



# setup内で vue-router

# Setup内でvue-router

setup内でthisは使えない。専用の合成関数を読み込む

<https://next.router.vuejs.org/guide/advanced/composition-api.html>

```
Import { useRouter, useRoute } from 'vue-router'
```

```
setup(){  
  const router = useRouter()  
  const route = useRoute()  
  
  const goHome = () => {  
    router.push('/')  
  }  
  const checkRoutePath = () => {  
    console.log(route.path)  
  }  
  
  return { goHome, checkRoutePath }  
}
```



# ナビゲーションガードも読み込む

```
import { onBeforeRouteLeave, onBeforeRouteUpdate } from 'vue-router'

setup(){

  onBeforeRouteLeave((to, from)=>{
    console.log(`to: ${to}`)
    console.log(`from: ${from}`)
  })

  onBeforeRouteUpdate((to, from)=>{

  })

  return {}
}
```



setup内でvuex

# setup内でvuex

Vuexも合成関数が用意されている  
<https://next.vuex.vuejs.org/>

```
Import { computed } from 'vue'  
Import { useStore } from 'vuex'
```

```
setup(){  
  const store = useStore()  
  const count = computed(()=>{  
    return store.state.count  
  })
```

```
  const increment = () => {  
    store.commit('increment')
```

```
  return { count, increment }  
}
```

# setup内でvuex store/index.js

store/index.js

(createStore以外は以前と同じ)

```
export default createStore({
  state: {
    Count: 0
  },
  mutations: {
    increment(state){
      state.count++
    }
  }
})
```