PART01

响应式核心

setup( )

在选项式 API 的组件中集成基于组合式 API 的代码时使用

```vue
<template>
  <h1>{{ count }}</h1>
</template>

<script>
import { ref } from "vue";
export default {
  setup() {
    const count = ref(0);
    // 返回的值会暴露给template和其他选项
    return { count };
  },
  created() {
    console.log(this.count);
  },
};
</script>
```

什么是响应性

	A	B	C
0	1		
1	2		
2	3		

这里单元格 A2 中的值是通过公式 = A0 + A1 来定义的，因此最终得到的值为 3，正如所料。但如果你试着更改 A0 或 A1，你会注意到 A2 也随即自动更新了。

而 JavaScript 默认并不是这样的。如果我们用 JavaScript 写类似的逻辑：

```
let A0 = 1
let A1 = 2
let A2 = A0 + A1

console.log(A2) // 3

A0 = 2
console.log(A2) // 仍然是 3
```

所以: 需要使用一些封装过的函数才可以实现响应式的效果：

reactive ( )

ref ( )

声明响应式状态 reactive ()

```vue
<template>
  <h1>{{obj1.count}}</h1>
  <h1>{{obj2.count}}</h1>
  <button @click="incre1">count+1</button>
  <button @click="incre2">count+1</button>
</template>

<script setup>
import {reactive} from 'vue'
// 普通对象
const obj1 = {count: 1}
// 响应式对象
const obj2 = reactive({count: 1})
// 修改普通对象里的count
function incre1() {
  obj1.count++ // 不会触发更新
}
// 修改响应式对象里的count
function incre2() {
  obj2.count++ // 会触发更新
}
</script>
```

reactive( )的限制

仅对对象类型有效：{ }, [ ]
对原始类型无效：string, number, boolean, null, undefined, symbol, bigint

声明响应式状态 ref ()

ref( ): 创建一个响应、可更改的ref对象，该对象有一个指向其内部值的属性 .value

```
<template>
  <h1>{{ number }}</h1>
</template>

<script setup>
import { ref } from "vue";
const number = ref(1);
</script>
```

ref 对象是可更改的，也就是说你可以为 .value 赋予新的值。

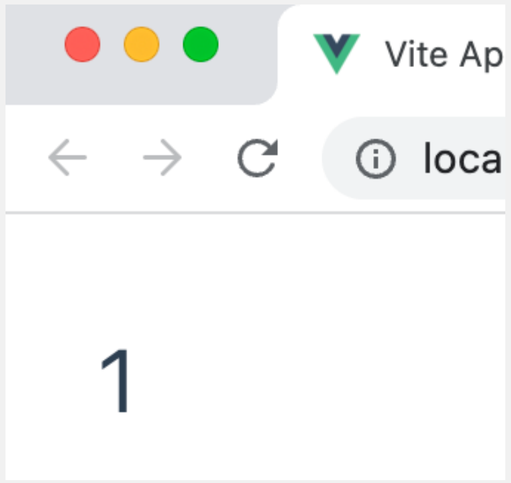
```
<template>
  <h1>{{ number }}</h1>
</template>

<script setup>
import { ref } from "vue";
let number = ref(1);
number.value++;
</script>
```

深层响应性

reactive()是深层响应式的。这意味着即使在更改深层次的对象或数组，你的改动也能被检测到。

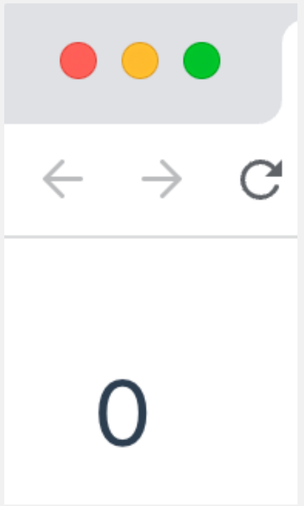
当obj.count的值被改变了，一样会被检测到，从而发生更新

```vue
<template>
  <h1>{{ obj.count }}</h1>
</template>

<script setup>
import { reactive } from "vue";
const obj = reactive({ count: 0 });
obj.count++;
</script>
```



但ref是浅层响应式

```vue
<template>
  <h1>{{ obj.count }}</h1>
</template>

<script setup>
import { ref } from "vue";
const obj = ref({ count: 0 });
obj.count++;
</script>
```



computed( ) 计算属性

```vue
<template>
  <h1>{{ doubleNumber }}</h1>
</template>

<script setup>
import { ref, computed } from "vue";
// 声明响应式属性
const number = ref(1);
// 声明计算属性
const doubleNumber = computed(() => {
  return number.value * 2;
});
</script>
```

readonly( ) 只读代理

接受一个对象 (不论是响应式还是普通的) 或是一个 ref，返回一个原值的只读代理。
下图示例中尝试修改copy将报错

```vue
<template>
  <h1>{{ copy.count }}</h1>
</template>

<script setup>
import { ref, readonly } from "vue";
const obj = ref({ count: 1 });
const copy = readonly(obj);
copy.count++;
</script>
```

只读代理是深层的：
对任何嵌套属性的访问都将是只读的。

避免深层级的转换行为，使用

shallowReadonly()

watch ( ) 侦听器

侦听一个或多个响应式数据源，并在数据源变化时调用所给的回调函数

watch()接收两个参数：
第一个参数是所侦听的数据源。
第二个参数是在发生变化时要调用的回调函数。

```
<template>
  <h1>{{ number }}</h1>
</template>

<script setup>
import { ref, watch } from "vue";
const number = ref(0);
watch(number, () => {
  console.log("变了");
});
number.value++;
</script>
```

watch ( ) 侦听器

侦听一个或多个响应式数据源，并在数据源变化时调用所给的回调函数

watch中的回调函数也可以接收参数：
第一个参数是新值
第二个参数是旧值

```
<template>
  <h1>{{ number }}</h1>
</template>

<script setup>
import { ref, watch } from "vue";
const number = ref(0);
watch(number, (newNumber, prevNumber) => {
  console.log("从" + prevNumber + "变成了" + newNumber);
});
number.value++;
</script>
```

PART02

响应式工具

ifRef ( )

检查某个值是否为 ref，返回值是一个类型判定。

```
<script setup>
import { ref, isRef } from "vue";
const number = ref(0);
console.log(isRef(number));
</script>
```

```
true
```

unref ( )

如果参数是 ref，则返回内部值，否则返回参数本身。

```
<script setup>
import { ref, unref } from "vue";
const number = ref(0);
const n = unref(number);
console.log(n == number.value); // true
console.log(n == number); // false
</script>
```

toRef ( )

把reactive对象中的一个属性创建为ref，改变源属性的值将更新 ref 的值，反之亦然

toRef的第一个参数是reactive对象，第二个参数是这个对象中的属性

```
<script setup>
import { reactive, toRef } from "vue";
const obj = reactive({ age: 10, name: "adam" });
const age = toRef(obj, "age");
console.log(age); // ref对象
console.log(age.value); // 10
</script>
```

isProxy ( )

检查一个对象是否是由 reactive()、readonly()、shallowReactive() 或 shallowReadonly() 创建的。

```
<script setup>
import { isProxy, reactive, ref, readonly } from "vue";
const arr = reactive([1, 2, 3]);
const num = ref(0);
const copy = readonly(arr);
console.log(isProxy(arr)); // true
console.log(isProxy(num)); // false
console.log(isProxy(copy)); // true
</script>
```

isReactive ( )

检查一个对象是否是由 reactive() 或 shallowReactive() 创建的

```
<script setup>
import { isReactive, reactive, ref } from "vue";
const arr = reactive([1, 2, 3]);
const num = ref(0);
console.log(isReactive(arr)); // true
console.log(isReactive(num)); // false
</script>
```

isReadonly ( )

检查传入的值是否为只读对象。

```
<script setup>
import { isReadonly, ref, readonly } from "vue";
const num = ref(0);
const copy = readonly(num);
console.log(isReadonly(num)); // false
console.log(isReadonly(copy)); // true
</script>
```

PART03

生命周期函数

生命周期函数

	选项式API	组合式API
创建前	beforeCreate	无， setup() 函数本身就是在组件实例创建之前执行的
创建后	created	无，同上
渲染前	beforeMount	onBeforeMount
渲染后	mounted	onMounted
更新前	beforeUpdate	onBeforeUpdate
更新后	updated	onUpdated
销毁前	beforeUnmount	onBeforeUnmount
销毁后	unmounted	onUnmounted

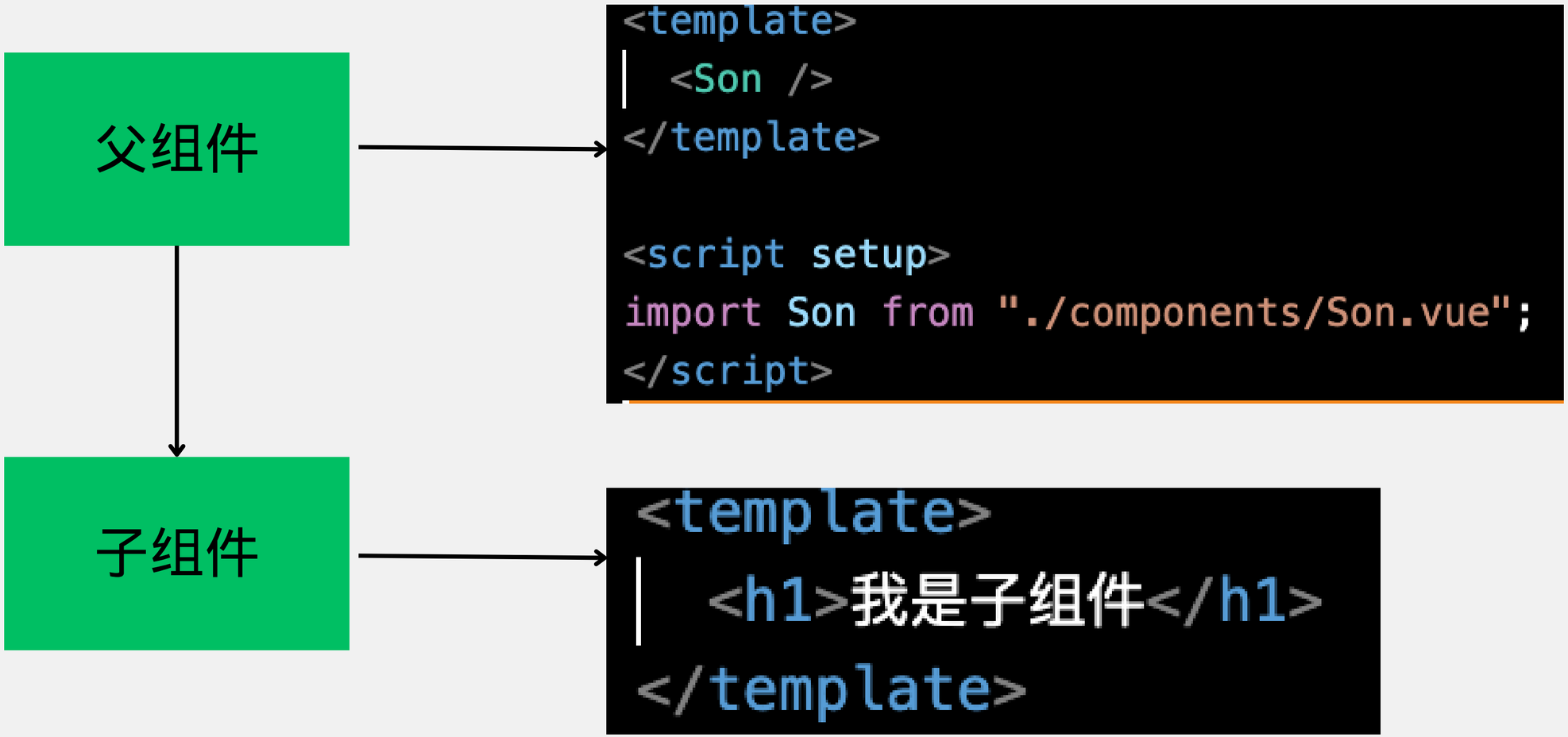
生命周期函数 onMounted示例

```vue
<script setup>
import { onMounted } from "vue";
onMounted(() => {
  console.log("123");
});
</script>
```

PART04

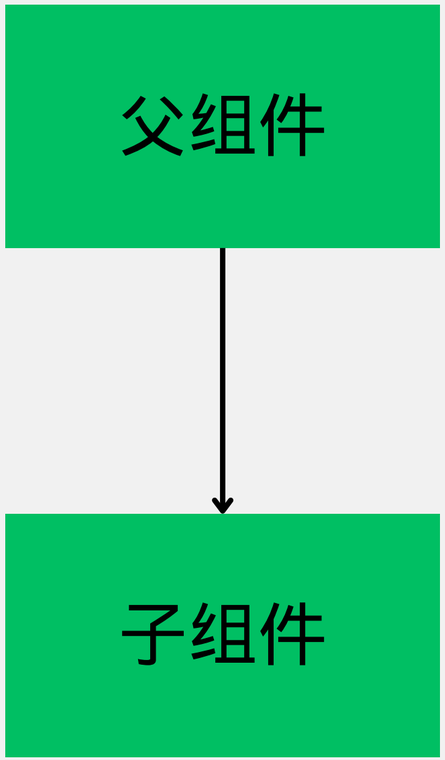
依赖注入

组件的局部注册


父组件
子组件
<template>
<Son />
</template>
<script setup>
import Son from "./components/Son.vue";
</script>
<template>
<h1>我是子组件</h1>
</template>

组件数据通信 - 依赖注入

组合式API的依赖注入主要用于在
Vue 组件之间共享代码和数据



provide (key, value)

inject (key)

```vue
<template>
  <Son />
</template>

<script setup>
import Son from "./components/Son.vue";
import { ref, provide } from "vue";

const number = ref(0);
// 父组件中提供一个值
provide("number", number);
</script>
```

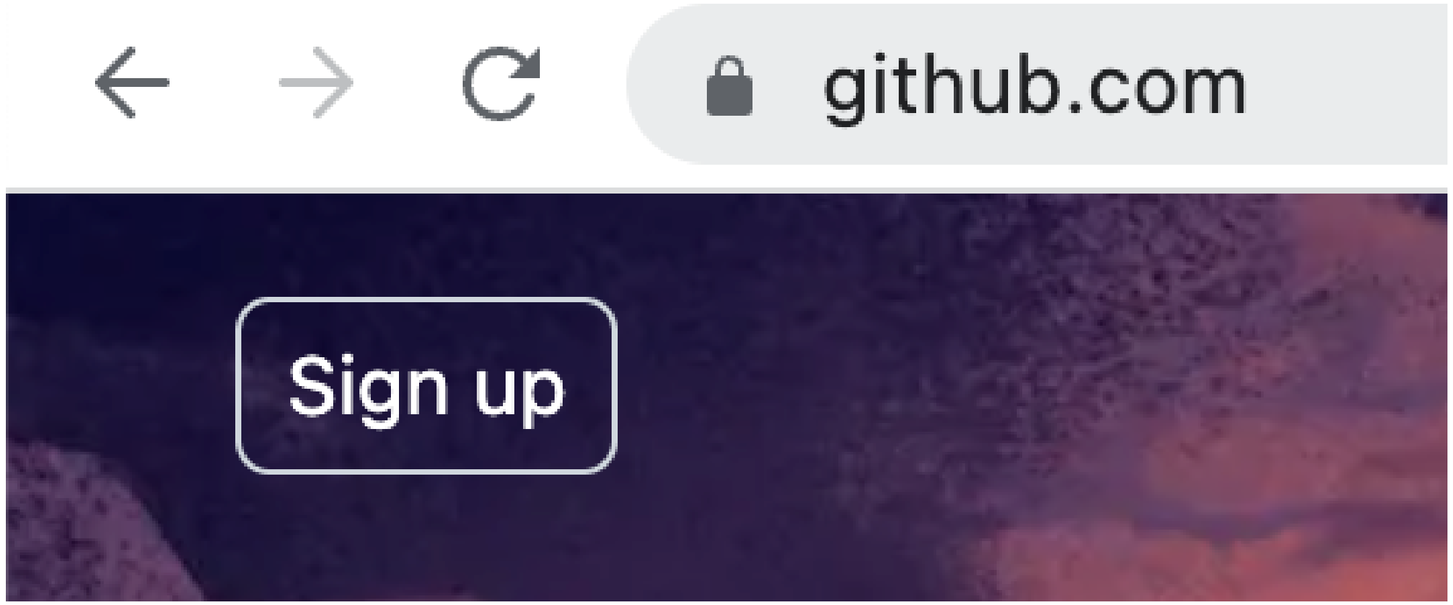
```vue
<template>
  <div>
    <h1>我是子组件</h1>
    <h1>{{ number }}</h1>
  </div>
</template>

<script setup>
import { inject } from "vue";
// 子组件中注入这个值
const number = inject("number");
</script>
```

PART05

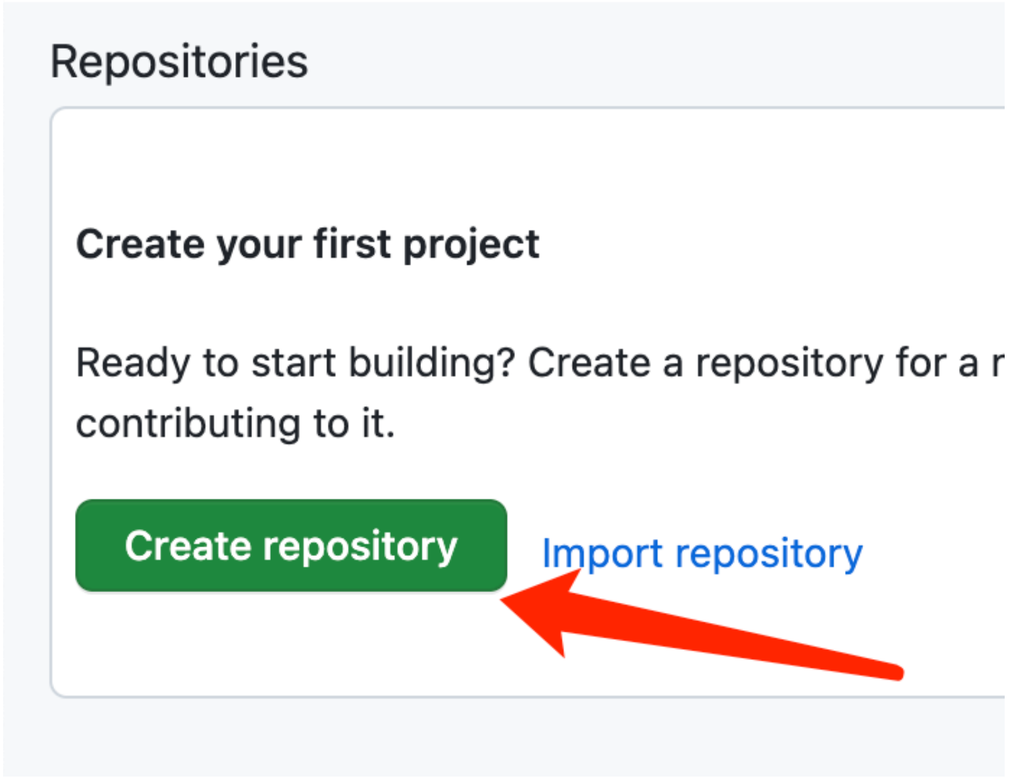
github pages

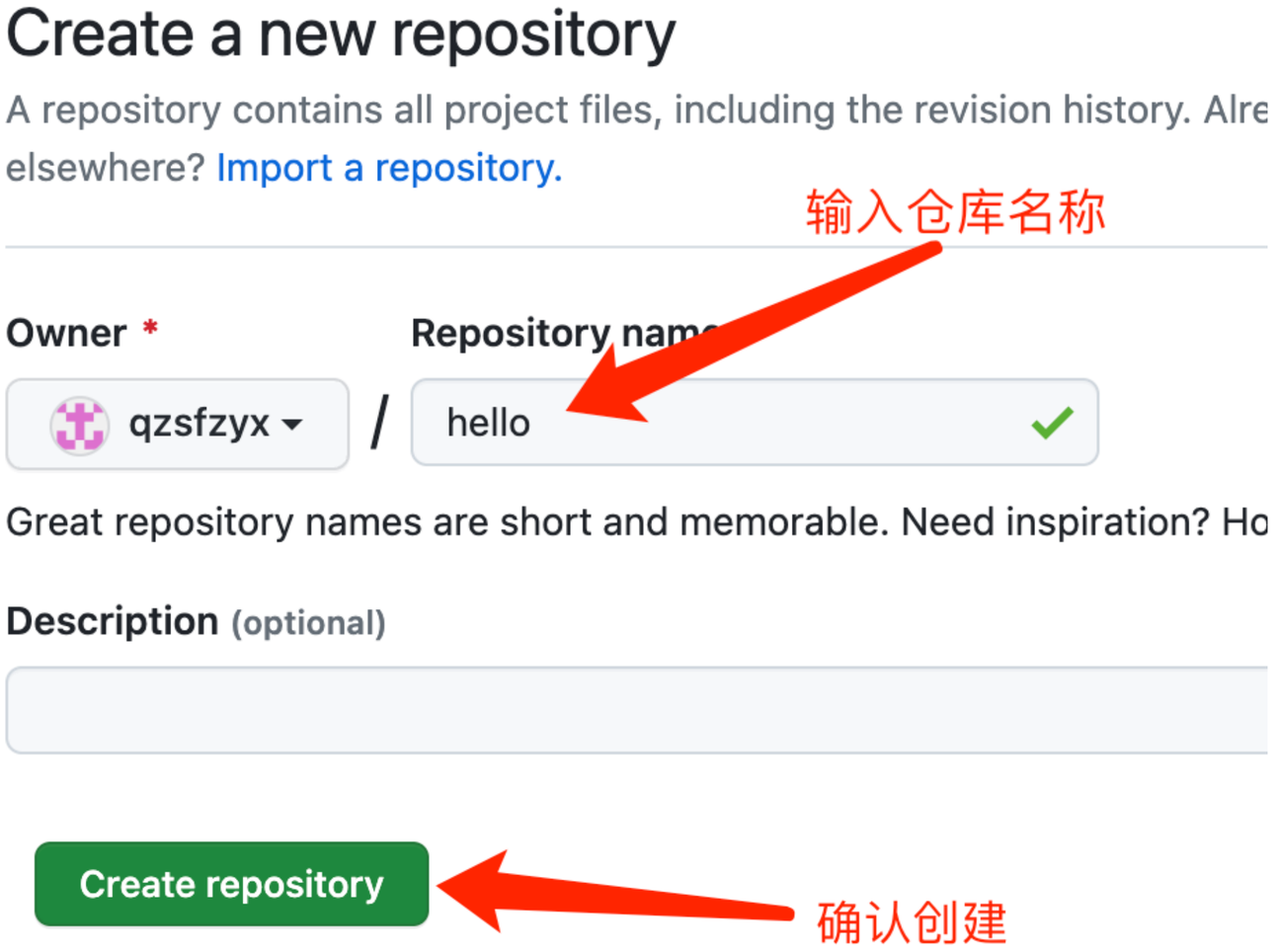
免费静态网站部署

创建github账号



创建仓库（repository）





打开vite.config.js，添加base设置

打包

npm run build

打包完毕会多出dist

将项目推送到远端仓库

进入目录	cd dist
创建本地git仓库	git init
添加到暂存区	git add .
提交到本地git仓库	git commit -m "deploy"
重命名分支	git branch -M gh-pages
推送到远端仓库	git push -u <远端url> gh-pages

把项目推到github - 创建一个git本地仓库

```
cd dist
git init
```

在当前目录（dist）中创建一个新的 Git 仓库

执行 git init 命令会在当前目录下创建一个名为 .git 的隐藏文件夹，它包含了 Git 用于管理版本控制的所有必要文件和目录。

注意：执行 git init 命令只是在当前目录中创建了一个新的 Git 仓库，它并没有将当前目录中的任何文件添加到版本控制中。需要使用 git add 命令将要管理的文件添加到 Git 仓库中，才能开始进行版本控制。

把项目推到github - 将dist添加到暂存区

git add .

用于将当前目录下所有尚未被 Git 跟踪的文件添加到 Git 仓库中，以便进行版本控制。

执行 git add . 命令会将当前目录下所有未被忽略的文件和子目录添加到 Git 的暂存区中，准备进行下一步的 git commit 操作

注意：git add . 命令不会将已经被 Git 跟踪的文件和目录重复添加到暂存区中，而只会将尚未被跟踪的新文件和目录添加到暂存区中。如果您想要添加指定的文件或目录到 Git 仓库中，可以使用 git add <文件名> 或 git add <文件夹名> 命令来完成。

把项目推到github - 将暂存区中的文件添加到仓库

git commit -m "deploy"

用于将当前 Git 仓库中的所有已经添加到暂存区的文件保存到版本历史记录中，并添加一个简短的提交信息，以便于后续查看版本变更的记录。

执行 git commit -m "deploy" 命令会将所有已经添加到暂存区的文件以及相关的提交信息保存到 Git 仓库中，并将其视为一个新的版本。这个提交信息一般包括一个简短的描述该版本的修改内容，以便于其他开发者了解该版本的变更。

-m 选项后面跟着一条用引号括起来的字符串，表示提交的简短描述，通常包括修改的内容、原因、影响等信息。

把项目推到github - 重命名分支

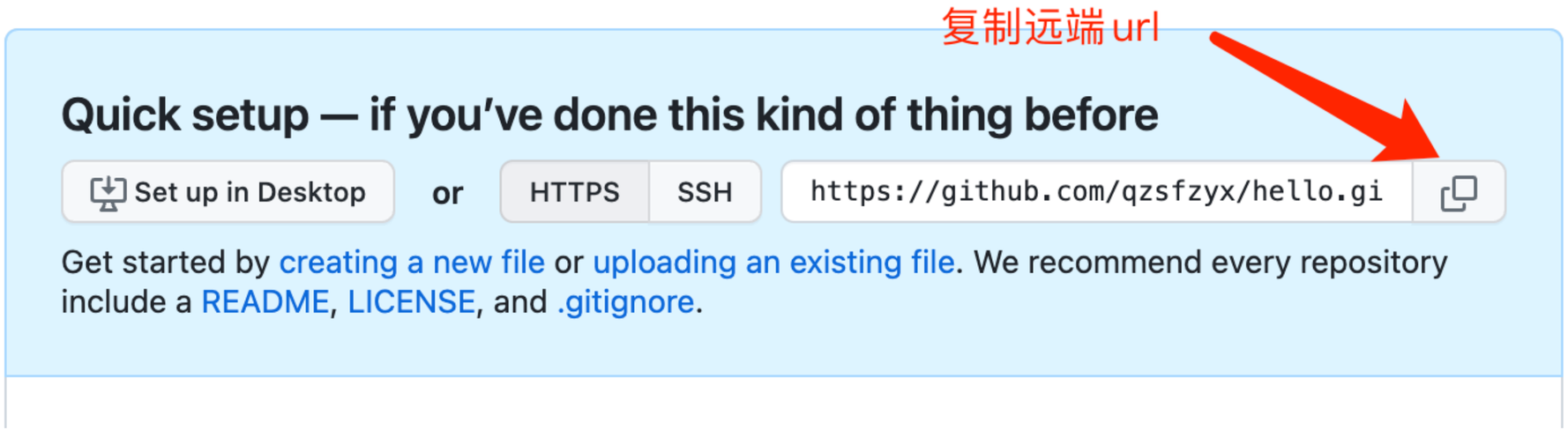
git branch -M gh-pages

用于重命名分支并强制更新。

执行 git branch -M gh-pages 命令会将当前分支重命名为 gh-pages，并且强制更新，以便与远程分支同步。如果 gh-pages 分支已经存在，则会覆盖原有的分支，否则就创建一个新的分支。

注意：使用 -M 选项时会强制更新分支，即使存在未合并的修改或者其他分支引用该分支，也会覆盖它们。因此，在使用 -M 选项之前，应该确保所有的修改都已经提交到 Git 仓库中，并且确保没有其他分支在引用该分支。

把项目推到github - 推送到远端仓库



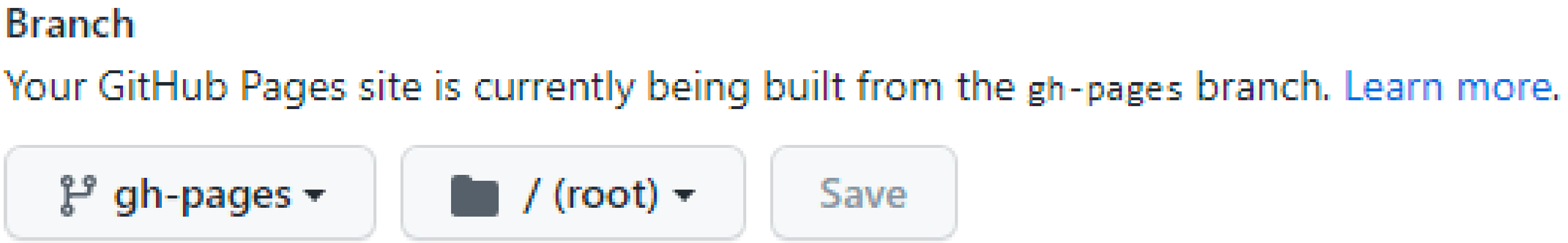
git push -u <远端url> gh-pages

用于将本地的 gh-pages 分支推送到指定的远程 Git 仓库中

-u 选项会将当前的分支与指定的远程仓库关联起来，并将该分支推送到远程仓库中。这样，在后续的推送操作中，可以直接使用 git push 命令将该分支的修改推送到远程仓库中。

设置

在github打开仓库（repository），点击Setting - Pages - Branch，选择gh-pages，点击save



等待10分钟左右即可

项目将上线在：

<github账号>.github.io/<仓库名称>