你好，我是winter。这一部分我们来讲一讲JavaScript的执行。

首先我们考虑一下，如果我们是浏览器或者Node的开发者，我们该如何使用JavaScript引擎。

当拿到一段JavaScript代码时，浏览器或者Node环境首先要做的就是：传递给JavaScript引擎，并且要求它去执行。

然而，执行JavaScript并非一锤子买卖，宿主环境当遇到一些事件时，会继续把一段代码传递给JavaScript引擎去执行，此外，我们可能还会提供API给JavaScript引擎，比如setTimeout这样的API，它会允许JavaScript在特定的时机执行。

所以，我们首先应该形成一个感性的认知：一个JavaScript引擎会常驻于内存中，它等待着我们（宿主）把JavaScript代码或者函数传递给它执行。

在ES3和更早的版本中，JavaScript本身还没有异步执行代码的能力，这也就意味着，宿主环境传递给JavaScript引擎一段代码，引擎就把代码直接顺次执行了，这个任务也就是宿主发起的任务。

但是，在ES5之后，JavaScript引入了Promise，这样，不需要浏览器的安排，JavaScript引擎本身也可以发起任务了。

由于我们这里主要讲JavaScript语言，那么采纳JSC引擎的术语，我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任务。

宏观和微观任务

JavaScript引擎等待宿主环境分配宏观任务，在操作系统中，通常等待的行为都是一个事件循环，所以在Node术语中，也会把这个部分称为事件循环。

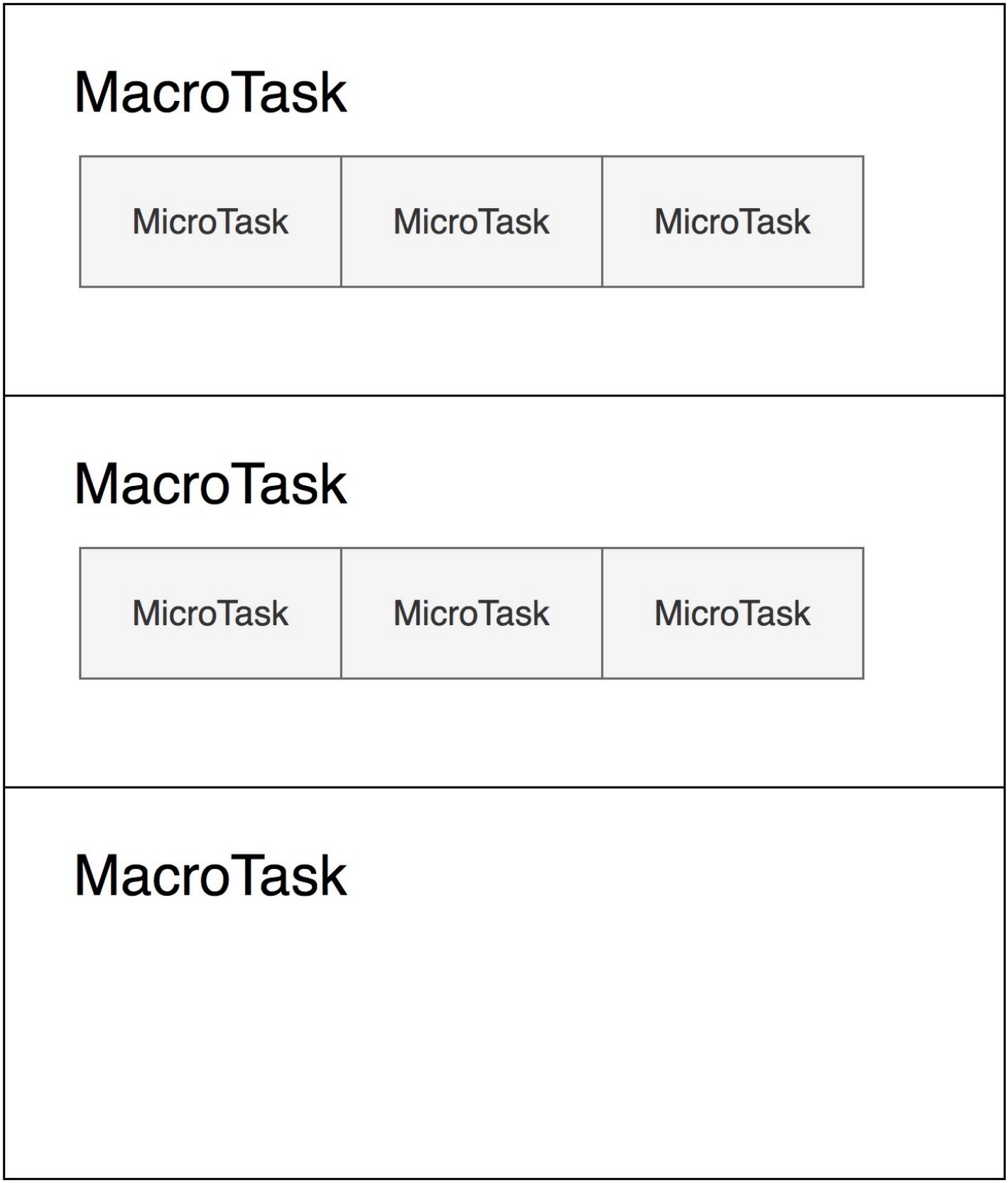
不过，术语本身并非我们需要重点讨论的内容，我们在这里把重点放在事件循环的原理上。在底层的C/C++代码中，这个事件循环是一个跑在独立线程中的循环，我们用伪代码来表示，大概是这样的：

```
while(TRUE) {
    r = wait();
    execute(r);
}
```

我们可以看到，整个循环做的事情基本上就是反复“等待-执行”。当然，实际的代码中并没有这么简单，还有要判断循环是否结束、宏观任务队列等逻辑，这里为了方便你理解，我就把这些都省略掉了。

这里每次的执行过程，其实都是一个宏观任务。我们可以大概理解：宏观任务的队列就相当于事件循环。

在宏观任务中，JavaScript的Promise还会产生异步代码，JavaScript必须保证这些异步代码在一个宏观任务中完成，因此，每个宏观任务中又包含了一个微观任务队列：



有了宏观任务和微观任务机制，我们就可以实现JavaScript引擎级和宿主级的任务了，例如：Promise永远在队列尾部添加微观任务。setTimeout等宿主API，则会添加宏观任务。

接下来，我们来详细介绍一下Promise。

Promise

Promise是JavaScript语言提供的一种标准化的异步管理方式，它的总体思想是，需要进行io、等待或者其它异步操作的函数，不返回真实结果，而返回一个“承诺”，函数的调用方可以在合适的时机，选择等待这个承诺兑现（通过Promise的then方法的回调）。

Promise的基本用法示例如下：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
sleep(1000).then( ()=> console.log("finished"));
```

这段代码定义了一个函数sleep，它的作用是等候传入参数指定的时长。

Promise的then回调是一个异步的执行过程，下面我们就来研究一下Promise函数中的执行顺序，我们来看一段代码示例：

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
r.then(() => console.log("c"));
console.log("b")
```

我们执行这段代码后，注意输出的顺序是 a b c。在进入console.log(“b”) 之前，毫无疑问 r 已经得到了resolve，但是Promise的resolve始终是异步操作，所以c无法出现在b之前。

接下来我们试试跟setTimeout混用的Promise。

在这段代码中，我设置了两段互不相干的异步操作：通过setTimeout执行console.log(“d”)，通过Promise执行console.log(“c”)。

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
setTimeout(()=>console.log("d"), 0)
r.then(() => console.log("c"));
console.log("b")
```

我们发现，不论代码顺序如何，d必定发生在c之后，因为Promise产生的是JavaScript引擎内部的微任务，而setTimeout是浏览器API，它产生宏任务。

为了理解微任务始终先于宏任务，我们设计一个实验：执行一个耗时1秒的Promise。

```
setTimeout(()=>console.log("d"), 0)
var r = new Promise(function(resolve, reject){
    resolve()
});
r.then(() => {
    var begin = Date.now();
    while(Date.now() - begin < 1000);
    console.log("c1")
    new Promise(function(resolve, reject){
        resolve()
    }).then(() => console.log("c2"))
});
```

这里我们强制了1秒的执行耗时，这样，我们可以确保任务c2是在d之后被添加到任务队列。

我们可以看到，即使耗时一秒的c1执行完毕，再enque的c2，仍然先于d执行了，这很好地解释了微任务优先的原理。

通过一系列的实验，我们可以总结一下如何分析异步执行的顺序：

- 首先我们分析有多少个宏任务；
- 在每个宏任务中，分析有多少个微任务；
- 根据调用次序，确定宏任务中的微任务执行次序；
- 根据宏任务的触发规则和调用次序，确定宏任务的执行次序；
- 确定整个顺序。

我们再来看一个稍微复杂的例子：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        console.log("b");
        setTimeout(resolve,duration);
    })
}
console.log("a");
sleep(5000).then(()=>console.log("c"));
```

这是一段非常常用的封装方法，利用Promise把setTimeout封装成可以用于异步的函数。

我们首先来看，setTimeout把整个代码分割成了2个宏观任务，这里不论是5秒还是0秒，都是一样的。

第一个宏观任务中，包含了先后同步执行的 console.log(“a”); 和 console.log(“b”);。

setTimeout后，第二个宏观任务执行调用了resolve，然后then中的代码异步得到执行，所以调用了console.log(“c”)，最终输出的顺序才是： a b c。

Promise是JavaScript中的一个定义，但是实际编写代码时，我们可以发现，它似乎并不比回调的方式书写更简单，但是从ES6开始，我们有了async/await，这个语法改进跟Promise配合，能够有效地改善代码结构。

新特性：async/await

async/await是ES2016新加入的特性，它提供了用for、if等代码结构来编写异步的方式。它的运行时基础是Promise，面对这种比较新的特性，我们先来看一下基本用法。

async函数必定返回Promise，我们把所有返回Promise的函数都可以认为是异步函数。

async函数是一种特殊语法，特征是在function关键字之前加上async关键字，这样，就定义了一个async函数，我们可以在其中使用await来等待一个Promise。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(){
    console.log("a")
    await sleep(2000)
    console.log("b")
}
```

这段代码利用了我们之前定义的sleep函数。在异步函数foo中，我们调用sleep。

async函数强大之处在于，它是可以嵌套的。我们在定义了一批原子操作的情况下，可以利用async函数组合出新的async函数。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(name){
    await sleep(2000)
    console.log(name)
}
async function foo2(){
    await foo("a");
    await foo("b");
}
```

这里foo2用await调用了两次异步函数foo，可以看到，如果我们把sleep这样的异步操作放入某一个框架或者库中，使用者几乎不需要了解Promise的概念即可进行异步编程了。

此外，generator/iterator也常常被跟异步一起来讲，我们必须说明 generator/iterator 并非异步代码，只是在缺少async/await的时候，一些框架（最著名的要数co）使用这样的特性来模拟async/await。

但是generator并非被设计成实现异步，所以有了async/await之后，generator/iterator来模拟异步的方法应该被废弃。

结语

在今天的文章里，我们学习了JavaScript执行部分的知识，首先我们学习了JavaScript的宏观任务和微观任务相关的知识。我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任

务。许多的微观任务的队列组成了宏观任务。

除此之外，我们还展开介绍了用Promise来添加微观任务的方式，并且介绍了async/await这个语法的改进。

最后，留给你一个小练习：我们现在要实现一个红绿灯，把一个圆形div按照绿色3秒，黄色1秒，红色2秒循环改变背景色，你会怎样编写这个代码呢？欢迎你留言讨论。

你好，我是winter。这一部分我们来讲一讲JavaScript的执行。

首先我们考虑一下，如果我们是浏览器或者Node的开发者，我们该如何使用JavaScript引擎。

当拿到一段JavaScript代码时，浏览器或者Node环境首先要做的就是：传递给JavaScript引擎，并且要求它去执行。

然而，执行JavaScript并非一锤子买卖，宿主环境当遇到一些事件时，会继续把一段代码传递给JavaScript引擎去执行，此外，我们可能还会提供API给JavaScript引擎，比如setTimeout这样的API，它会允许JavaScript在特定的时机执行。

所以，我们首先应该形成一个感性的认知：一个JavaScript引擎会常驻于内存中，它等待着我们（宿主）把JavaScript代码或者函数传递给它执行。

在ES3和更早的版本中，JavaScript本身还没有异步执行代码的能力，这也就意味着，宿主环境传递给JavaScript引擎一段代码，引擎就把代码直接顺次执行了，这个任务也就是宿主发起的任务。

但是，在ES5之后，JavaScript引入了Promise，这样，不需要浏览器的安排，JavaScript引擎本身也可以发起任务了。

由于我们这里主要讲JavaScript语言，那么采纳JSC引擎的术语，我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任务。

宏观和微观任务

JavaScript引擎等待宿主环境分配宏观任务，在操作系统中，通常等待的行为都是一个事件循环，所以在Node术语中，也会把这个部分称为事件循环。

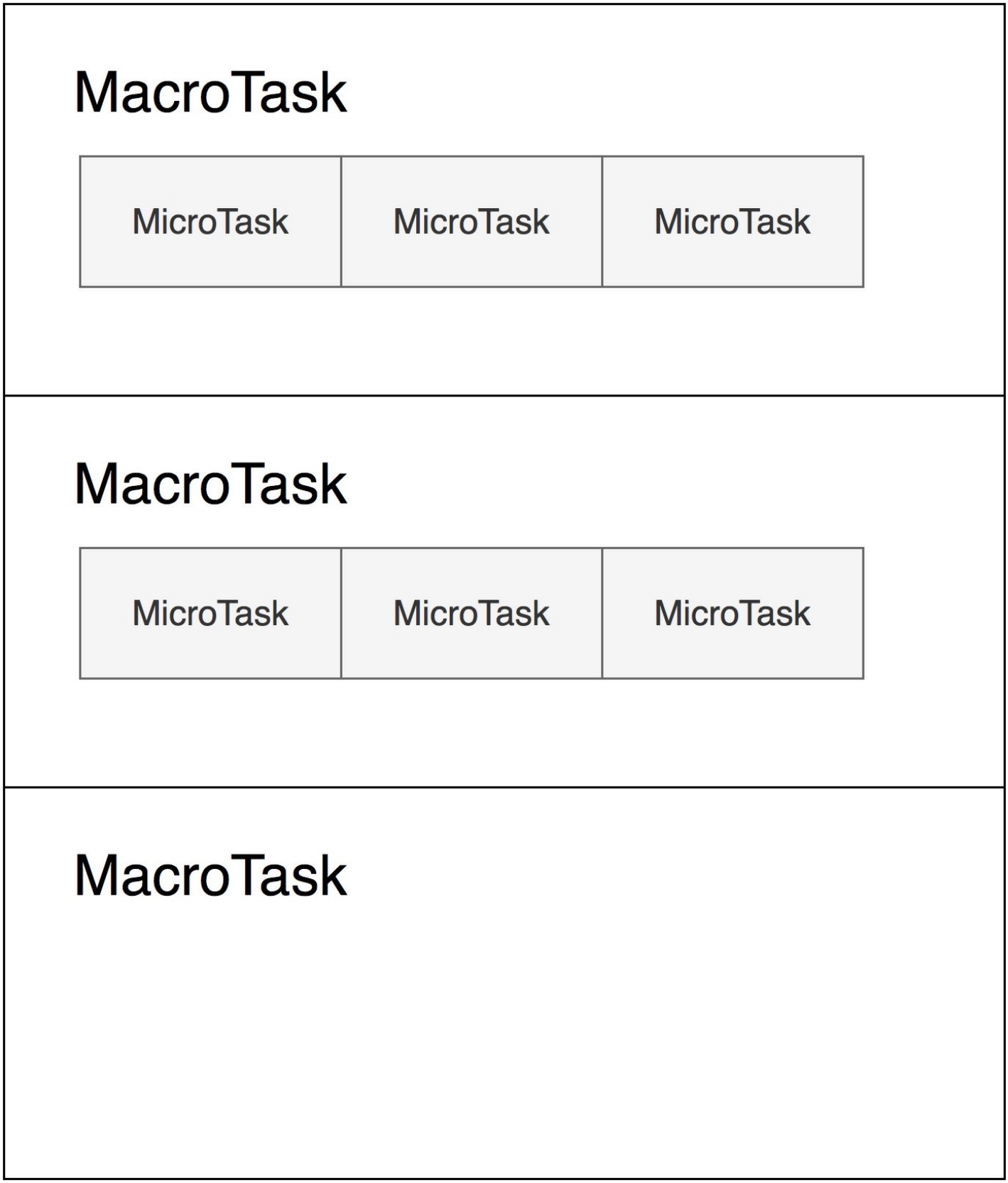
不过，术语本身并非我们需要重点讨论的内容，我们在这里把重点放在事件循环的原理上。在底层的C/C++代码中，这个事件循环是一个跑在独立线程中的循环，我们用伪代码来表示，大概是这样的：

```
while(TRUE) {
    r = wait();
    execute(r);
}
```

我们可以看到，整个循环做的事情基本上就是反复“等待-执行”。当然，实际的代码中并没有这么简单，还有要判断循环是否结束、宏观任务队列等逻辑，这里为了方便你理解，我就把这些都省略掉了。

这里每次的执行过程，其实都是一个宏观任务。我们可以大概理解：宏观任务的队列就相当于事件循环。

在宏观任务中，JavaScript的Promise还会产生异步代码，JavaScript必须保证这些异步代码在一个宏观任务中完成，因此，每个宏观任务中又包含了一个微观任务队列：



有了宏观任务和微观任务机制，我们就可以实现JavaScript引擎级和宿主级的任务了，例如：Promise永远在队列尾部添加微观任务。setTimeout等宿主API，则会添加宏观任务。

接下来，我们来详细介绍一下Promise。

Promise

Promise是JavaScript语言提供的一种标准化的异步管理方式，它的总体思想是，需要进行io、等待或者其它异步操作的函数，不返回真实结果，而返回一个“承诺”，函数的调用方可以在合适的时机，选择等待这个承诺兑现（通过Promise的then方法的回调）。

Promise的基本用法示例如下：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
sleep(1000).then( ()=> console.log("finished"));
```

这段代码定义了一个函数sleep，它的作用是等候传入参数指定的时长。

Promise的then回调是一个异步的执行过程，下面我们就来研究一下Promise函数中的执行顺序，我们来看一段代码示例：

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
r.then(() => console.log("c"));
console.log("b")
```

我们执行这段代码后，注意输出的顺序是 a b c。在进入console.log(“b”) 之前，毫无疑问 r 已经得到了resolve，但是Promise的resolve始终是异步操作，所以c无法出现在b之前。

接下来我们试试跟setTimeout混用的Promise。

在这段代码中，我设置了两段互不相干的异步操作：通过setTimeout执行console.log(“d”)，通过Promise执行console.log(“c”)。

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
setTimeout(()=>console.log("d"), 0)
r.then(() => console.log("c"));
console.log("b")
```

我们发现，不论代码顺序如何，d必定发生在c之后，因为Promise产生的是JavaScript引擎内部的微任务，而setTimeout是浏览器API，它产生宏任务。

为了理解微任务始终先于宏任务，我们设计一个实验：执行一个耗时1秒的Promise。

```
setTimeout(()=>console.log("d"), 0)
var r = new Promise(function(resolve, reject){
    resolve()
});
r.then(() => {
    var begin = Date.now();
    while(Date.now() - begin < 1000);
    console.log("c1")
    new Promise(function(resolve, reject){
        resolve()
    }).then(() => console.log("c2"))
});
```

这里我们强制了1秒的执行耗时，这样，我们可以确保任务c2是在d之后被添加到任务队列。

我们可以看到，即使耗时一秒的c1执行完毕，再enque的c2，仍然先于d执行了，这很好地解释了微任务优先的原理。

通过一系列的实验，我们可以总结一下如何分析异步执行的顺序：

- 首先我们分析有多少个宏任务；
- 在每个宏任务中，分析有多少个微任务；
- 根据调用次序，确定宏任务中的微任务执行次序；
- 根据宏任务的触发规则和调用次序，确定宏任务的执行次序；
- 确定整个顺序。

我们再来看一个稍微复杂的例子：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        console.log("b");
        setTimeout(resolve,duration);
    })
}
console.log("a");
sleep(5000).then(()=>console.log("c"));
```

这是一段非常常用的封装方法，利用Promise把setTimeout封装成可以用于异步的函数。

我们首先来看，setTimeout把整个代码分割成了2个宏观任务，这里不论是5秒还是0秒，都是一样的。

第一个宏观任务中，包含了先后同步执行的 console.log(“a”); 和 console.log(“b”);。

setTimeout后，第二个宏观任务执行调用了resolve，然后then中的代码异步得到执行，所以调用了console.log(“c”)，最终输出的顺序才是： a b c。

Promise是JavaScript中的一个定义，但是实际编写代码时，我们可以发现，它似乎并不比回调的方式书写更简单，但是从ES6开始，我们有了async/await，这个语法改进跟Promise配合，能够有效地改善代码结构。

新特性：async/await

async/await是ES2016新加入的特性，它提供了用for、if等代码结构来编写异步的方式。它的运行时基础是Promise，面对这种比较新的特性，我们先来看一下基本用法。

async函数必定返回Promise，我们把所有返回Promise的函数都可以认为是异步函数。

async函数是一种特殊语法，特征是在function关键字之前加上async关键字，这样，就定义了一个async函数，我们可以在其中使用await来等待一个Promise。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(){
    console.log("a")
    await sleep(2000)
    console.log("b")
}
```

这段代码利用了我们之前定义的sleep函数。在异步函数foo中，我们调用sleep。

async函数强大之处在于，它是可以嵌套的。我们在定义了一批原子操作的情况下，可以利用async函数组合出新的async函数。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(name){
    await sleep(2000)
    console.log(name)
}
async function foo2(){
    await foo("a");
    await foo("b");
}
```

这里foo2用await调用了两次异步函数foo，可以看到，如果我们把sleep这样的异步操作放入某一个框架或者库中，使用者几乎不需要了解Promise的概念即可进行异步编程了。

此外，generator/iterator也常常被跟异步一起来讲，我们必须说明 generator/iterator 并非异步代码，只是在缺少async/await的时候，一些框架（最著名的要数co）使用这样的特性来模拟async/await。

但是generator并非被设计成实现异步，所以有了async/await之后，generator/iterator来模拟异步的方法应该被废弃。

结语

在今天的文章里，我们学习了JavaScript执行部分的知识，首先我们学习了JavaScript的宏观任务和微观任务相关的知识。我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任

务。许多的微观任务的队列组成了宏观任务。

除此之外，我们还展开介绍了用Promise来添加微观任务的方式，并且介绍了async/await这个语法的改进。

最后，留给你一个小练习：我们现在要实现一个红绿灯，把一个圆形div按照绿色3秒，黄色1秒，红色2秒循环改变背景色，你会怎样编写这个代码呢？欢迎你留言讨论。

你好，我是winter。这一部分我们来讲一讲JavaScript的执行。

首先我们考虑一下，如果我们是浏览器或者Node的开发者，我们该如何使用JavaScript引擎。

当拿到一段JavaScript代码时，浏览器或者Node环境首先要做的就是：传递给JavaScript引擎，并且要求它去执行。

然而，执行JavaScript并非一锤子买卖，宿主环境当遇到一些事件时，会继续把一段代码传递给JavaScript引擎去执行，此外，我们可能还会提供API给JavaScript引擎，比如setTimeout这样的API，它会允许JavaScript在特定的时机执行。

所以，我们首先应该形成一个感性的认知：一个JavaScript引擎会常驻于内存中，它等待着我们（宿主）把JavaScript代码或者函数传递给它执行。

在ES3和更早的版本中，JavaScript本身还没有异步执行代码的能力，这也就意味着，宿主环境传递给JavaScript引擎一段代码，引擎就把代码直接顺次执行了，这个任务也就是宿主发起的任务。

但是，在ES5之后，JavaScript引入了Promise，这样，不需要浏览器的安排，JavaScript引擎本身也可以发起任务了。

由于我们这里主要讲JavaScript语言，那么采纳JSC引擎的术语，我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任务。

宏观和微观任务

JavaScript引擎等待宿主环境分配宏观任务，在操作系统中，通常等待的行为都是一个事件循环，所以在Node术语中，也会把这个部分称为事件循环。

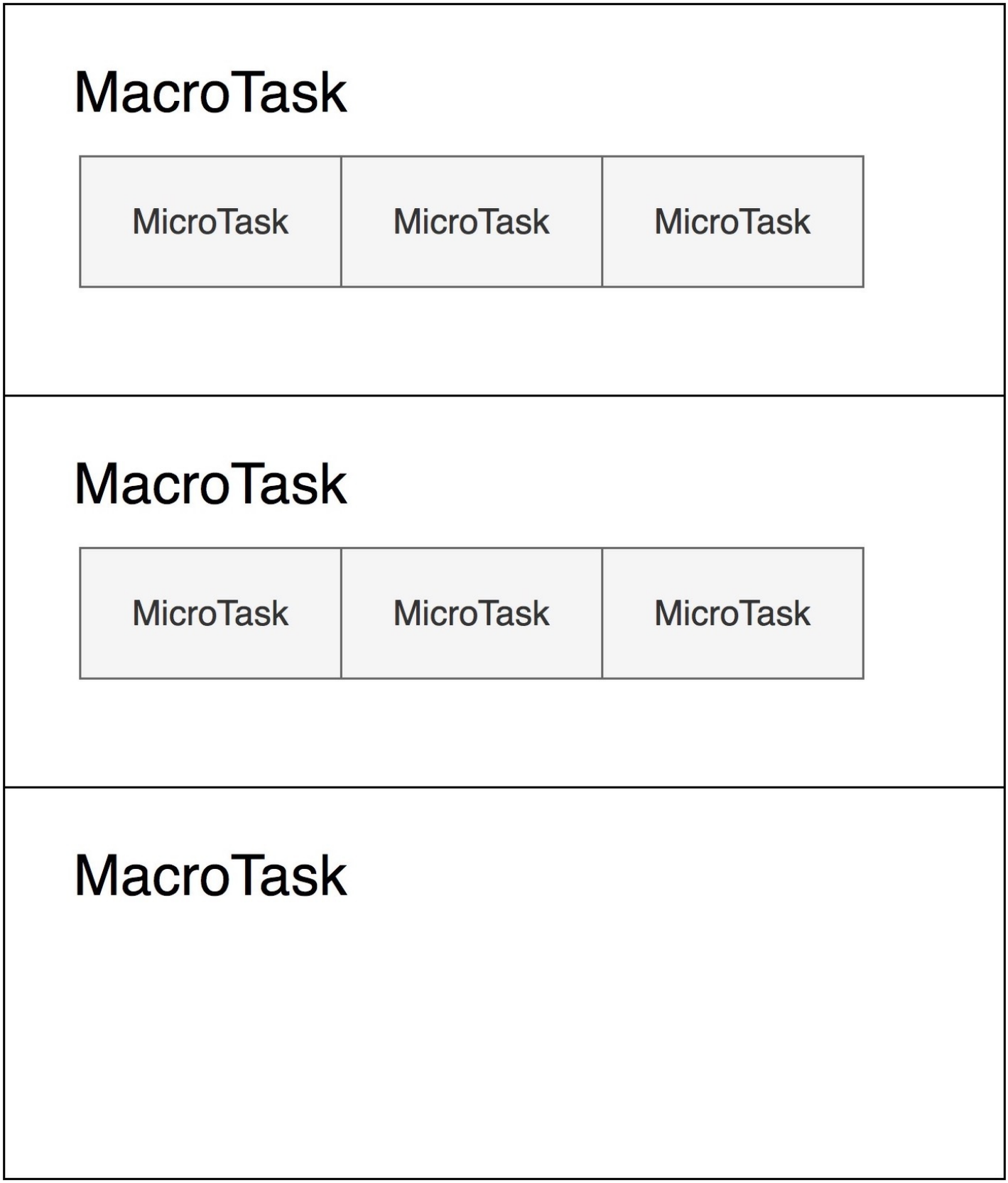
不过，术语本身并非我们需要重点讨论的内容，我们在这里把重点放在事件循环的原理上。在底层的C/C++代码中，这个事件循环是一个跑在独立线程中的循环，我们用伪代码来表示，大概是这样的：

```
while(TRUE) {
    r = wait();
    execute(r);
}
```

我们可以看到，整个循环做的事情基本上就是反复“等待-执行”。当然，实际的代码中并没有这么简单，还有要判断循环是否结束、宏观任务队列等逻辑，这里为了方便你理解，我就把这些都省略掉了。

这里每次的执行过程，其实都是一个宏观任务。我们可以大概理解：宏观任务的队列就相当于事件循环。

在宏观任务中，JavaScript的Promise还会产生异步代码，JavaScript必须保证这些异步代码在一个宏观任务中完成，因此，每个宏观任务中又包含了一个微观任务队列：



有了宏观任务和微观任务机制，我们就可以实现JavaScript引擎级和宿主级的任务了，例如：Promise永远在队列尾部添加微观任务。setTimeout等宿主API，则会添加宏观任务。

接下来，我们来详细介绍一下Promise。

Promise

Promise是JavaScript语言提供的一种标准化的异步管理方式，它的总体思想是，需要进行io、等待或者其它异步操作的函数，不返回真实结果，而返回一个“承诺”，函数的调用方可以在合适的时机，选择等待这个承诺兑现（通过Promise的then方法的回调）。

Promise的基本用法示例如下：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
sleep(1000).then( ()=> console.log("finished"));
```

这段代码定义了一个函数sleep，它的作用是等候传入参数指定的时长。

Promise的then回调是一个异步的执行过程，下面我们就来研究一下Promise函数中的执行顺序，我们来看一段代码示例：

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
r.then(() => console.log("c"));
console.log("b")
```

我们执行这段代码后，注意输出的顺序是 a b c。在进入console.log(“b”) 之前，毫无疑问 r 已经得到了resolve，但是Promise的resolve始终是异步操作，所以c无法出现在b之前。

接下来我们试试跟setTimeout混用的Promise。

在这段代码中，我设置了两段互不相干的异步操作：通过setTimeout执行console.log(“d”)，通过Promise执行console.log(“c”)。

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
setTimeout(()=>console.log("d"), 0)
r.then(() => console.log("c"));
console.log("b")
```

我们发现，不论代码顺序如何，d必定发生在c之后，因为Promise产生的是JavaScript引擎内部的微任务，而setTimeout是浏览器API，它产生宏任务。

为了理解微任务始终先于宏任务，我们设计一个实验：执行一个耗时1秒的Promise。

```
setTimeout(()=>console.log("d"), 0)
var r = new Promise(function(resolve, reject){
    resolve()
});
r.then(() => {
    var begin = Date.now();
    while(Date.now() - begin < 1000);
    console.log("c1")
    new Promise(function(resolve, reject){
        resolve()
    }).then(() => console.log("c2"))
});
```

这里我们强制了1秒的执行耗时，这样，我们可以确保任务c2是在d之后被添加到任务队列。

我们可以看到，即使耗时一秒的c1执行完毕，再enque的c2，仍然先于d执行了，这很好地解释了微任务优先的原理。

通过一系列的实验，我们可以总结一下如何分析异步执行的顺序：

- 首先我们分析有多少个宏任务；
- 在每个宏任务中，分析有多少个微任务；
- 根据调用次序，确定宏任务中的微任务执行次序；
- 根据宏任务的触发规则和调用次序，确定宏任务的执行次序；
- 确定整个顺序。

我们再来看一个稍微复杂的例子：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        console.log("b");
        setTimeout(resolve,duration);
    })
}
console.log("a");
sleep(5000).then(()=>console.log("c"));
```

这是一段非常常用的封装方法，利用Promise把setTimeout封装成可以用于异步的函数。

我们首先来看，setTimeout把整个代码分割成了2个宏观任务，这里不论是5秒还是0秒，都是一样的。

第一个宏观任务中，包含了先后同步执行的 console.log(“a”); 和 console.log(“b”);。

setTimeout后，第二个宏观任务执行调用了resolve，然后then中的代码异步得到执行，所以调用了console.log(“c”)，最终输出的顺序才是： a b c。

Promise是JavaScript中的一个定义，但是实际编写代码时，我们可以发现，它似乎并不比回调的方式书写更简单，但是从ES6开始，我们有了async/await，这个语法改进跟Promise配合，能够有效地改善代码结构。

新特性：async/await

async/await是ES2016新加入的特性，它提供了用for、if等代码结构来编写异步的方式。它的运行时基础是Promise，面对这种比较新的特性，我们先来看一下基本用法。

async函数必定返回Promise，我们把所有返回Promise的函数都可以认为是异步函数。

async函数是一种特殊语法，特征是在function关键字之前加上async关键字，这样，就定义了一个async函数，我们可以在其中使用await来等待一个Promise。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(){
    console.log("a")
    await sleep(2000)
    console.log("b")
}
```

这段代码利用了我们之前定义的sleep函数。在异步函数foo中，我们调用sleep。

async函数强大之处在于，它是可以嵌套的。我们在定义了一批原子操作的情况下，可以利用async函数组合出新的async函数。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(name){
    await sleep(2000)
    console.log(name)
}
async function foo2(){
    await foo("a");
    await foo("b");
}
```

这里foo2用await调用了两次异步函数foo，可以看到，如果我们把sleep这样的异步操作放入某一个框架或者库中，使用者几乎不需要了解Promise的概念即可进行异步编程了。

此外，generator/iterator也常常被跟异步一起来讲，我们必须说明 generator/iterator 并非异步代码，只是在缺少async/await的时候，一些框架（最著名的要数co）使用这样的特性来模拟async/await。

但是generator并非被设计成实现异步，所以有了async/await之后，generator/iterator来模拟异步的方法应该被废弃。

结语

在今天的文章里，我们学习了JavaScript执行部分的知识，首先我们学习了JavaScript的宏观任务和微观任务相关的知识。我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任

务。许多的微观任务的队列组成了宏观任务。

除此之外，我们还展开介绍了用Promise来添加微观任务的方式，并且介绍了async/await这个语法的改进。

最后，留给你一个小练习：我们现在要实现一个红绿灯，把一个圆形div按照绿色3秒，黄色1秒，红色2秒循环改变背景色，你会怎样编写这个代码呢？欢迎你留言讨论。

你好，我是winter。这一部分我们来讲一讲JavaScript的执行。

首先我们考虑一下，如果我们是浏览器或者Node的开发者，我们该如何使用JavaScript引擎。

当拿到一段JavaScript代码时，浏览器或者Node环境首先要做的就是：传递给JavaScript引擎，并且要求它去执行。

然而，执行JavaScript并非一锤子买卖，宿主环境当遇到一些事件时，会继续把一段代码传递给JavaScript引擎去执行，此外，我们可能还会提供API给JavaScript引擎，比如setTimeout这样的API，它会允许JavaScript在特定的时机执行。

所以，我们首先应该形成一个感性的认知：一个JavaScript引擎会常驻于内存中，它等待着我们（宿主）把JavaScript代码或者函数传递给它执行。

在ES3和更早的版本中，JavaScript本身还没有异步执行代码的能力，这也就意味着，宿主环境传递给JavaScript引擎一段代码，引擎就把代码直接顺次执行了，这个任务也就是宿主发起的任务。

但是，在ES5之后，JavaScript引入了Promise，这样，不需要浏览器的安排，JavaScript引擎本身也可以发起任务了。

由于我们这里主要讲JavaScript语言，那么采纳JSC引擎的术语，我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任务。

宏观和微观任务

JavaScript引擎等待宿主环境分配宏观任务，在操作系统中，通常等待的行为都是一个事件循环，所以在Node术语中，也会把这个部分称为事件循环。

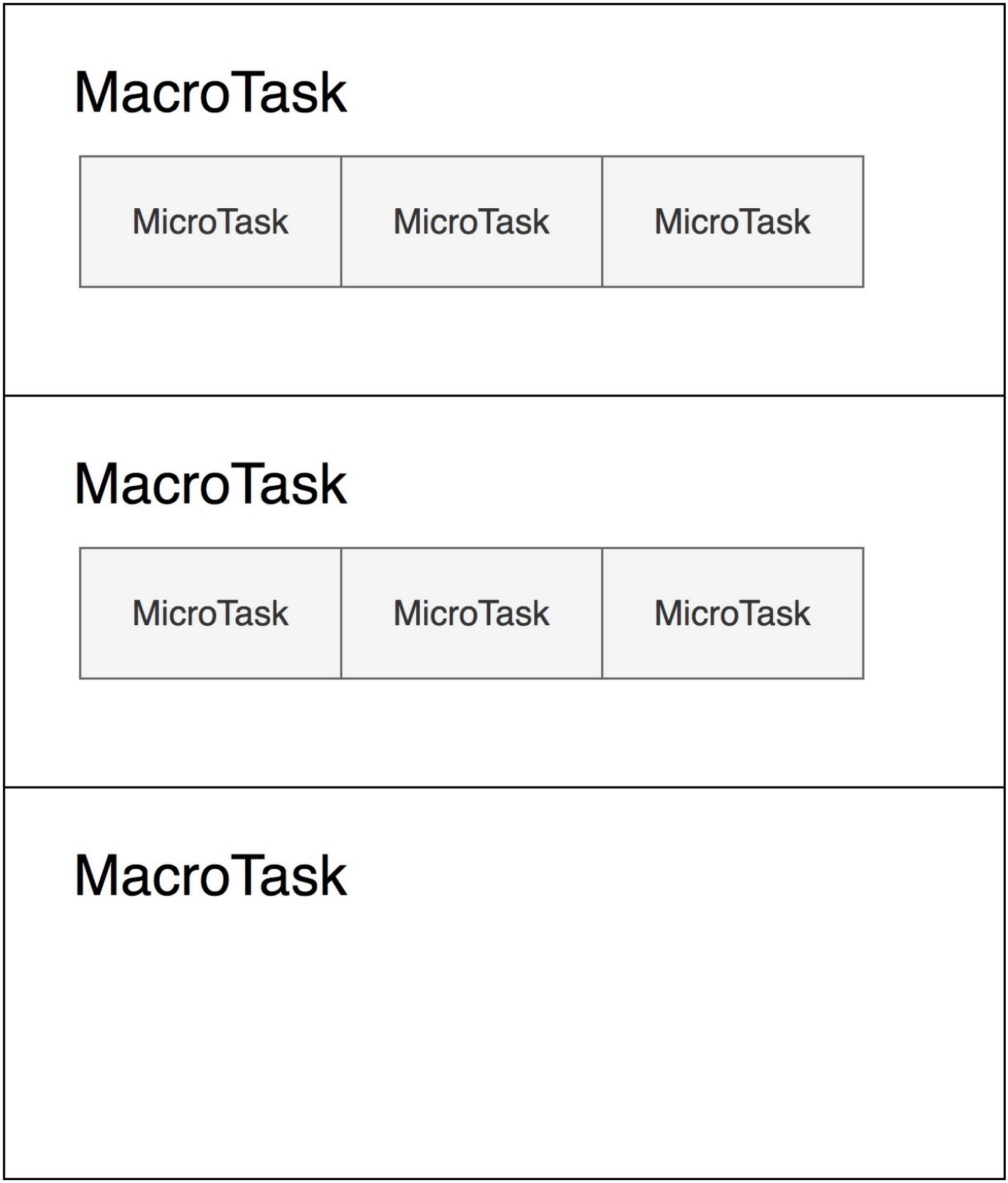
不过，术语本身并非我们需要重点讨论的内容，我们在这里把重点放在事件循环的原理上。在底层的C/C++代码中，这个事件循环是一个跑在独立线程中的循环，我们用伪代码来表示，大概是这样的：

```
while(TRUE) {
    r = wait();
    execute(r);
}
```

我们可以看到，整个循环做的事情基本上就是反复“等待-执行”。当然，实际的代码中并没有这么简单，还有要判断循环是否结束、宏观任务队列等逻辑，这里为了方便你理解，我就把这些都省略掉了。

这里每次的执行过程，其实都是一个宏观任务。我们可以大概理解：宏观任务的队列就相当于事件循环。

在宏观任务中，JavaScript的Promise还会产生异步代码，JavaScript必须保证这些异步代码在一个宏观任务中完成，因此，每个宏观任务中又包含了一个微观任务队列：



有了宏观任务和微观任务机制，我们就可以实现JavaScript引擎级和宿主级的任务了，例如：Promise永远在队列尾部添加微观任务。setTimeout等宿主API，则会添加宏观任务。

接下来，我们来详细介绍一下Promise。

Promise

Promise是JavaScript语言提供的一种标准化的异步管理方式，它的总体思想是，需要进行io、等待或者其它异步操作的函数，不返回真实结果，而返回一个“承诺”，函数的调用方可以在合适的时机，选择等待这个承诺兑现（通过Promise的then方法的回调）。

Promise的基本用法示例如下：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
sleep(1000).then( ()=> console.log("finished"));
```

这段代码定义了一个函数sleep，它的作用是等候传入参数指定的时长。

Promise的then回调是一个异步的执行过程，下面我们就来研究一下Promise函数中的执行顺序，我们来看一段代码示例：

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
r.then(() => console.log("c"));
console.log("b")
```

我们执行这段代码后，注意输出的顺序是 a b c。在进入console.log(“b”) 之前，毫无疑问 r 已经得到了resolve，但是Promise的resolve始终是异步操作，所以c无法出现在b之前。

接下来我们试试跟setTimeout混用的Promise。

在这段代码中，我设置了两段互不相干的异步操作：通过setTimeout执行console.log(“d”)，通过Promise执行console.log(“c”)。

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
setTimeout(()=>console.log("d"), 0)
r.then(() => console.log("c"));
console.log("b")
```

我们发现，不论代码顺序如何，d必定发生在c之后，因为Promise产生的是JavaScript引擎内部的微任务，而setTimeout是浏览器API，它产生宏任务。

为了理解微任务始终先于宏任务，我们设计一个实验：执行一个耗时1秒的Promise。

```
setTimeout(()=>console.log("d"), 0)
var r = new Promise(function(resolve, reject){
    resolve()
});
r.then(() => {
    var begin = Date.now();
    while(Date.now() - begin < 1000);
    console.log("c1")
    new Promise(function(resolve, reject){
        resolve()
    }).then(() => console.log("c2"))
});
```

这里我们强制了1秒的执行耗时，这样，我们可以确保任务c2是在d之后被添加到任务队列。

我们可以看到，即使耗时一秒的c1执行完毕，再enque的c2，仍然先于d执行了，这很好地解释了微任务优先的原理。

通过一系列的实验，我们可以总结一下如何分析异步执行的顺序：

- 首先我们分析有多少个宏任务；
- 在每个宏任务中，分析有多少个微任务；
- 根据调用次序，确定宏任务中的微任务执行次序；
- 根据宏任务的触发规则和调用次序，确定宏任务的执行次序；
- 确定整个顺序。

我们再来看一个稍微复杂的例子：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        console.log("b");
        setTimeout(resolve,duration);
    })
}
console.log("a");
sleep(5000).then(()=>console.log("c"));
```

这是一段非常常用的封装方法，利用Promise把setTimeout封装成可以用于异步的函数。

我们首先来看，setTimeout把整个代码分割成了2个宏观任务，这里不论是5秒还是0秒，都是一样的。

第一个宏观任务中，包含了先后同步执行的 console.log(“a”); 和 console.log(“b”);。

setTimeout后，第二个宏观任务执行调用了resolve，然后then中的代码异步得到执行，所以调用了console.log(“c”)，最终输出的顺序才是： a b c。

Promise是JavaScript中的一个定义，但是实际编写代码时，我们可以发现，它似乎并不比回调的方式书写更简单，但是从ES6开始，我们有了async/await，这个语法改进跟Promise配合，能够有效地改善代码结构。

新特性：async/await

async/await是ES2016新加入的特性，它提供了用for、if等代码结构来编写异步的方式。它的运行时基础是Promise，面对这种比较新的特性，我们先来看一下基本用法。

async函数必定返回Promise，我们把所有返回Promise的函数都可以认为是异步函数。

async函数是一种特殊语法，特征是在function关键字之前加上async关键字，这样，就定义了一个async函数，我们可以在其中使用await来等待一个Promise。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(){
    console.log("a")
    await sleep(2000)
    console.log("b")
}
```

这段代码利用了我们之前定义的sleep函数。在异步函数foo中，我们调用sleep。

async函数强大之处在于，它是可以嵌套的。我们在定义了一批原子操作的情况下，可以利用async函数组合出新的async函数。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(name){
    await sleep(2000)
    console.log(name)
}
async function foo2(){
    await foo("a");
    await foo("b");
}
```

这里foo2用await调用了两次异步函数foo，可以看到，如果我们把sleep这样的异步操作放入某一个框架或者库中，使用者几乎不需要了解Promise的概念即可进行异步编程了。

此外，generator/iterator也常常被跟异步一起来讲，我们必须说明 generator/iterator 并非异步代码，只是在缺少async/await的时候，一些框架（最著名的要数co）使用这样的特性来模拟async/await。

但是generator并非被设计成实现异步，所以有了async/await之后，generator/iterator来模拟异步的方法应该被废弃。

结语

在今天的文章里，我们学习了JavaScript执行部分的知识，首先我们学习了JavaScript的宏观任务和微观任务相关的知识。我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任

务。许多的微观任务的队列组成了宏观任务。

除此之外，我们还展开介绍了用Promise来添加微观任务的方式，并且介绍了async/await这个语法的改进。

最后，留给你一个小练习：我们现在要实现一个红绿灯，把一个圆形div按照绿色3秒，黄色1秒，红色2秒循环改变背景色，你会怎样编写这个代码呢？欢迎你留言讨论。

你好，我是winter。这一部分我们来讲一讲JavaScript的执行。

首先我们考虑一下，如果我们是浏览器或者Node的开发者，我们该如何使用JavaScript引擎。

当拿到一段JavaScript代码时，浏览器或者Node环境首先要做的就是：传递给JavaScript引擎，并且要求它去执行。

然而，执行JavaScript并非一锤子买卖，宿主环境当遇到一些事件时，会继续把一段代码传递给JavaScript引擎去执行，此外，我们可能还会提供API给JavaScript引擎，比如setTimeout这样的API，它会允许JavaScript在特定的时机执行。

所以，我们首先应该形成一个感性的认知：一个JavaScript引擎会常驻于内存中，它等待着我们（宿主）把JavaScript代码或者函数传递给它执行。

在ES3和更早的版本中，JavaScript本身还没有异步执行代码的能力，这也就意味着，宿主环境传递给JavaScript引擎一段代码，引擎就把代码直接顺次执行了，这个任务也就是宿主发起的任务。

但是，在ES5之后，JavaScript引入了Promise，这样，不需要浏览器的安排，JavaScript引擎本身也可以发起任务了。

由于我们这里主要讲JavaScript语言，那么采纳JSC引擎的术语，我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任务。

宏观和微观任务

JavaScript引擎等待宿主环境分配宏观任务，在操作系统中，通常等待的行为都是一个事件循环，所以在Node术语中，也会把这个部分称为事件循环。

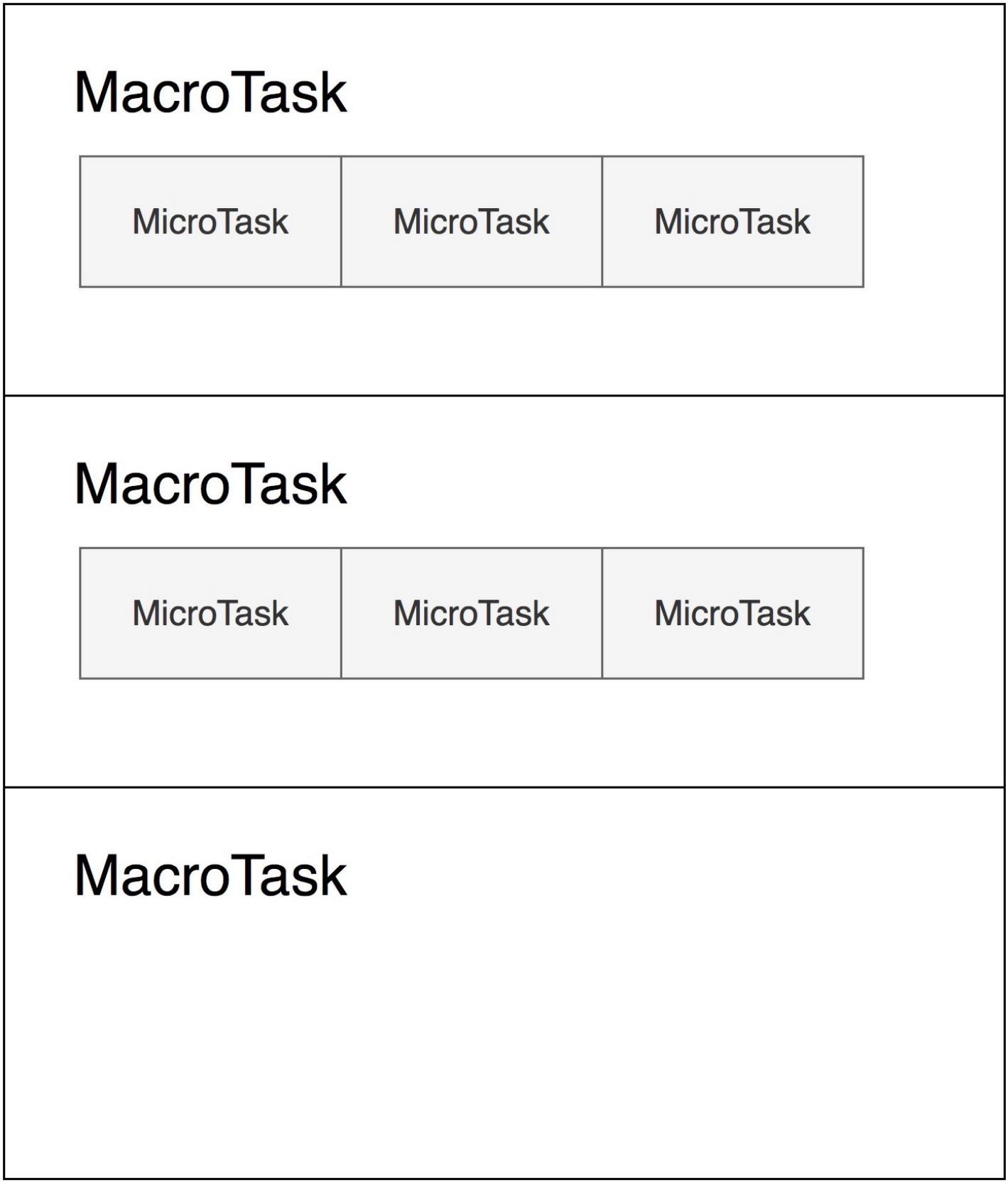
不过，术语本身并非我们需要重点讨论的内容，我们在这里把重点放在事件循环的原理上。在底层的C/C++代码中，这个事件循环是一个跑在独立线程中的循环，我们用伪代码来表示，大概是这样的：

```
while(TRUE) {
    r = wait();
    execute(r);
}
```

我们可以看到，整个循环做的事情基本上就是反复“等待-执行”。当然，实际的代码中并没有这么简单，还有要判断循环是否结束、宏观任务队列等逻辑，这里为了方便你理解，我就把这些都省略掉了。

这里每次的执行过程，其实都是一个宏观任务。我们可以大概理解：宏观任务的队列就相当于事件循环。

在宏观任务中，JavaScript的Promise还会产生异步代码，JavaScript必须保证这些异步代码在一个宏观任务中完成，因此，每个宏观任务中又包含了一个微观任务队列：



有了宏观任务和微观任务机制，我们就可以实现JavaScript引擎级和宿主级的任务了，例如：Promise永远在队列尾部添加微观任务。setTimeout等宿主API，则会添加宏观任务。

接下来，我们来详细介绍一下Promise。

Promise

Promise是JavaScript语言提供的一种标准化的异步管理方式，它的总体思想是，需要进行io、等待或者其它异步操作的函数，不返回真实结果，而返回一个“承诺”，函数的调用方可以在合适的时机，选择等待这个承诺兑现（通过Promise的then方法的回调）。

Promise的基本用法示例如下：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
sleep(1000).then( ()=> console.log("finished"));
```

这段代码定义了一个函数sleep，它的作用是等候传入参数指定的时长。

Promise的then回调是一个异步的执行过程，下面我们就来研究一下Promise函数中的执行顺序，我们来看一段代码示例：

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
r.then(() => console.log("c"));
console.log("b")
```

我们执行这段代码后，注意输出的顺序是 a b c。在进入console.log(“b”) 之前，毫无疑问 r 已经得到了resolve，但是Promise的resolve始终是异步操作，所以c无法出现在b之前。

接下来我们试试跟setTimeout混用的Promise。

在这段代码中，我设置了两段互不相干的异步操作：通过setTimeout执行console.log(“d”)，通过Promise执行console.log(“c”)。

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
setTimeout(()=>console.log("d"), 0)
r.then(() => console.log("c"));
console.log("b")
```

我们发现，不论代码顺序如何，d必定发生在c之后，因为Promise产生的是JavaScript引擎内部的微任务，而setTimeout是浏览器API，它产生宏任务。

为了理解微任务始终先于宏任务，我们设计一个实验：执行一个耗时1秒的Promise。

```
setTimeout(()=>console.log("d"), 0)
var r = new Promise(function(resolve, reject){
    resolve()
});
r.then(() => {
    var begin = Date.now();
    while(Date.now() - begin < 1000);
    console.log("c1")
    new Promise(function(resolve, reject){
        resolve()
    }).then(() => console.log("c2"))
});
```

这里我们强制了1秒的执行耗时，这样，我们可以确保任务c2是在d之后被添加到任务队列。

我们可以看到，即使耗时一秒的c1执行完毕，再enque的c2，仍然先于d执行了，这很好地解释了微任务优先的原理。

通过一系列的实验，我们可以总结一下如何分析异步执行的顺序：

- 首先我们分析有多少个宏任务；
- 在每个宏任务中，分析有多少个微任务；
- 根据调用次序，确定宏任务中的微任务执行次序；
- 根据宏任务的触发规则和调用次序，确定宏任务的执行次序；
- 确定整个顺序。

我们再来看一个稍微复杂的例子：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        console.log("b");
        setTimeout(resolve,duration);
    })
}
console.log("a");
sleep(5000).then(()=>console.log("c"));
```

这是一段非常常用的封装方法，利用Promise把setTimeout封装成可以用于异步的函数。

我们首先来看，setTimeout把整个代码分割成了2个宏观任务，这里不论是5秒还是0秒，都是一样的。

第一个宏观任务中，包含了先后同步执行的 console.log(“a”); 和 console.log(“b”);。

setTimeout后，第二个宏观任务执行调用了resolve，然后then中的代码异步得到执行，所以调用了console.log(“c”)，最终输出的顺序才是： a b c。

Promise是JavaScript中的一个定义，但是实际编写代码时，我们可以发现，它似乎并不比回调的方式书写更简单，但是从ES6开始，我们有了async/await，这个语法改进跟Promise配合，能够有效地改善代码结构。

新特性：async/await

async/await是ES2016新加入的特性，它提供了用for、if等代码结构来编写异步的方式。它的运行时基础是Promise，面对这种比较新的特性，我们先来看一下基本用法。

async函数必定返回Promise，我们把所有返回Promise的函数都可以认为是异步函数。

async函数是一种特殊语法，特征是在function关键字之前加上async关键字，这样，就定义了一个async函数，我们可以在其中使用await来等待一个Promise。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(){
    console.log("a")
    await sleep(2000)
    console.log("b")
}
```

这段代码利用了我们之前定义的sleep函数。在异步函数foo中，我们调用sleep。

async函数强大之处在于，它是可以嵌套的。我们在定义了一批原子操作的情况下，可以利用async函数组合出新的async函数。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(name){
    await sleep(2000)
    console.log(name)
}
async function foo2(){
    await foo("a");
    await foo("b");
}
```

这里foo2用await调用了两次异步函数foo，可以看到，如果我们把sleep这样的异步操作放入某一个框架或者库中，使用者几乎不需要了解Promise的概念即可进行异步编程了。

此外，generator/iterator也常常被跟异步一起来讲，我们必须说明 generator/iterator 并非异步代码，只是在缺少async/await的时候，一些框架（最著名的要数co）使用这样的特性来模拟async/await。

但是generator并非被设计成实现异步，所以有了async/await之后，generator/iterator来模拟异步的方法应该被废弃。

结语

在今天的文章里，我们学习了JavaScript执行部分的知识，首先我们学习了JavaScript的宏观任务和微观任务相关的知识。我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任

务。许多的微观任务的队列组成了宏观任务。

除此之外，我们还展开介绍了用Promise来添加微观任务的方式，并且介绍了async/await这个语法的改进。

最后，留给你一个小练习：我们现在要实现一个红绿灯，把一个圆形div按照绿色3秒，黄色1秒，红色2秒循环改变背景色，你会怎样编写这个代码呢？欢迎你留言讨论。

你好，我是winter。这一部分我们来讲一讲JavaScript的执行。

首先我们考虑一下，如果我们是浏览器或者Node的开发者，我们该如何使用JavaScript引擎。

当拿到一段JavaScript代码时，浏览器或者Node环境首先要做的就是：传递给JavaScript引擎，并且要求它去执行。

然而，执行JavaScript并非一锤子买卖，宿主环境当遇到一些事件时，会继续把一段代码传递给JavaScript引擎去执行，此外，我们可能还会提供API给JavaScript引擎，比如setTimeout这样的API，它会允许JavaScript在特定的时机执行。

所以，我们首先应该形成一个感性的认知：一个JavaScript引擎会常驻于内存中，它等待着我们（宿主）把JavaScript代码或者函数传递给它执行。

在ES3和更早的版本中，JavaScript本身还没有异步执行代码的能力，这也就意味着，宿主环境传递给JavaScript引擎一段代码，引擎就把代码直接顺次执行了，这个任务也就是宿主发起的任务。

但是，在ES5之后，JavaScript引入了Promise，这样，不需要浏览器的安排，JavaScript引擎本身也可以发起任务了。

由于我们这里主要讲JavaScript语言，那么采纳JSC引擎的术语，我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任务。

宏观和微观任务

JavaScript引擎等待宿主环境分配宏观任务，在操作系统中，通常等待的行为都是一个事件循环，所以在Node术语中，也会把这个部分称为事件循环。

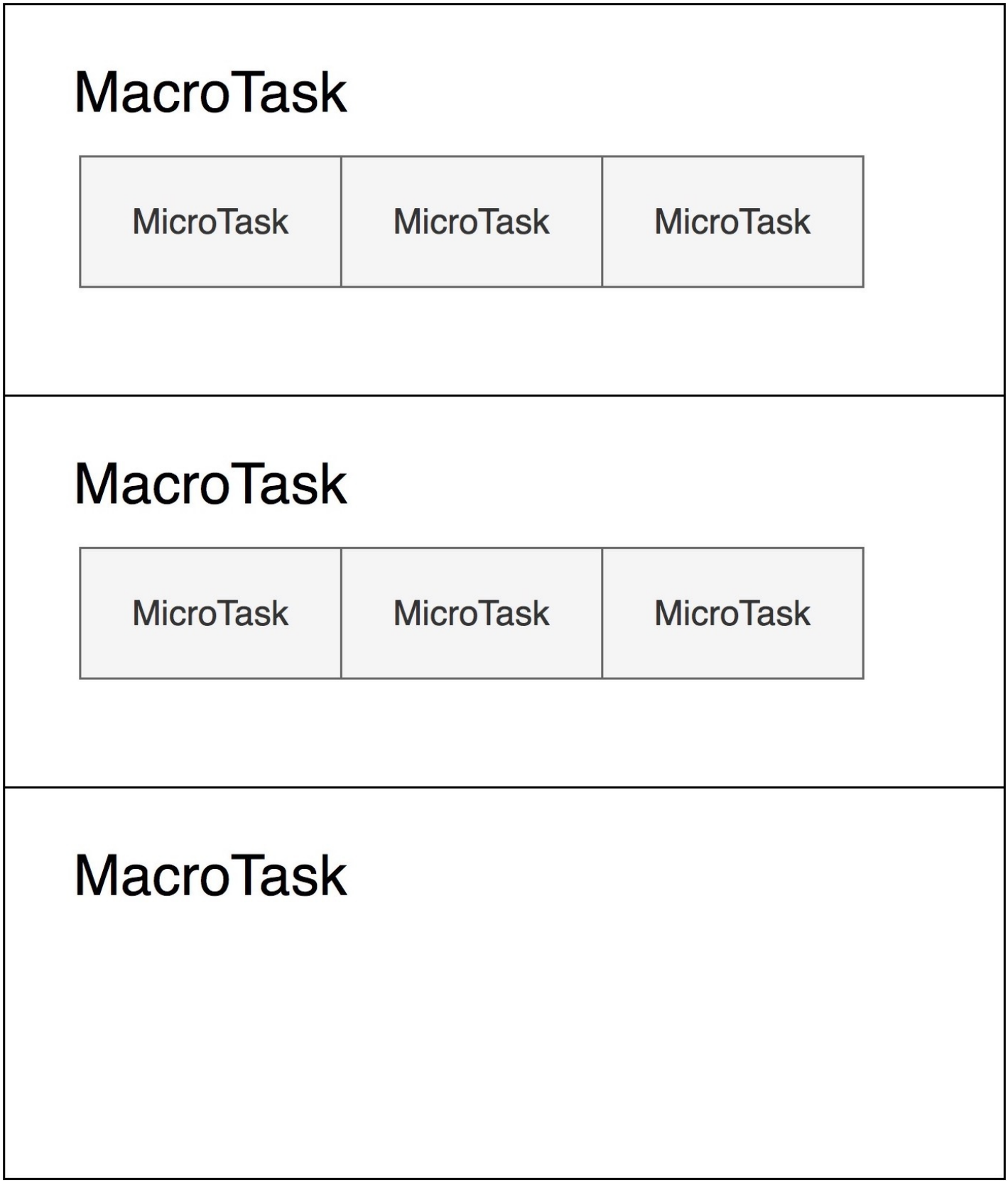
不过，术语本身并非我们需要重点讨论的内容，我们在这里把重点放在事件循环的原理上。在底层的C/C++代码中，这个事件循环是一个跑在独立线程中的循环，我们用伪代码来表示，大概是这样的：

```
while(TRUE) {
    r = wait();
    execute(r);
}
```

我们可以看到，整个循环做的事情基本上就是反复“等待-执行”。当然，实际的代码中并没有这么简单，还有要判断循环是否结束、宏观任务队列等逻辑，这里为了方便你理解，我就把这些都省略掉了。

这里每次的执行过程，其实都是一个宏观任务。我们可以大概理解：宏观任务的队列就相当于事件循环。

在宏观任务中，JavaScript的Promise还会产生异步代码，JavaScript必须保证这些异步代码在一个宏观任务中完成，因此，每个宏观任务中又包含了一个微观任务队列：



有了宏观任务和微观任务机制，我们就可以实现JavaScript引擎级和宿主级的任务了，例如：Promise永远在队列尾部添加微观任务。setTimeout等宿主API，则会添加宏观任务。

接下来，我们来详细介绍一下Promise。

Promise

Promise是JavaScript语言提供的一种标准化的异步管理方式，它的总体思想是，需要进行io、等待或者其它异步操作的函数，不返回真实结果，而返回一个“承诺”，函数的调用方可以在合适的时机，选择等待这个承诺兑现（通过Promise的then方法的回调）。

Promise的基本用法示例如下：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
sleep(1000).then( ()=> console.log("finished"));
```

这段代码定义了一个函数sleep，它的作用是等候传入参数指定的时长。

Promise的then回调是一个异步的执行过程，下面我们就来研究一下Promise函数中的执行顺序，我们来看一段代码示例：

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
r.then(() => console.log("c"));
console.log("b")
```

我们执行这段代码后，注意输出的顺序是 a b c。在进入console.log(“b”) 之前，毫无疑问 r 已经得到了resolve，但是Promise的resolve始终是异步操作，所以c无法出现在b之前。

接下来我们试试跟setTimeout混用的Promise。

在这段代码中，我设置了两段互不相干的异步操作：通过setTimeout执行console.log(“d”)，通过Promise执行console.log(“c”)。

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
setTimeout(()=>console.log("d"), 0)
r.then(() => console.log("c"));
console.log("b")
```

我们发现，不论代码顺序如何，d必定发生在c之后，因为Promise产生的是JavaScript引擎内部的微任务，而setTimeout是浏览器API，它产生宏任务。

为了理解微任务始终先于宏任务，我们设计一个实验：执行一个耗时1秒的Promise。

```
setTimeout(()=>console.log("d"), 0)
var r = new Promise(function(resolve, reject){
    resolve()
});
r.then(() => {
    var begin = Date.now();
    while(Date.now() - begin < 1000);
    console.log("c1")
    new Promise(function(resolve, reject){
        resolve()
    }).then(() => console.log("c2"))
});
```

这里我们强制了1秒的执行耗时，这样，我们可以确保任务c2是在d之后被添加到任务队列。

我们可以看到，即使耗时一秒的c1执行完毕，再enque的c2，仍然先于d执行了，这很好地解释了微任务优先的原理。

通过一系列的实验，我们可以总结一下如何分析异步执行的顺序：

- 首先我们分析有多少个宏任务；
- 在每个宏任务中，分析有多少个微任务；
- 根据调用次序，确定宏任务中的微任务执行次序；
- 根据宏任务的触发规则和调用次序，确定宏任务的执行次序；
- 确定整个顺序。

我们再来看一个稍微复杂的例子：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        console.log("b");
        setTimeout(resolve,duration);
    })
}
console.log("a");
sleep(5000).then(()=>console.log("c"));
```

这是一段非常常用的封装方法，利用Promise把setTimeout封装成可以用于异步的函数。

我们首先来看，setTimeout把整个代码分割成了2个宏观任务，这里不论是5秒还是0秒，都是一样的。

第一个宏观任务中，包含了先后同步执行的 console.log(“a”); 和 console.log(“b”);。

setTimeout后，第二个宏观任务执行调用了resolve，然后then中的代码异步得到执行，所以调用了console.log(“c”)，最终输出的顺序才是： a b c。

Promise是JavaScript中的一个定义，但是实际编写代码时，我们可以发现，它似乎并不比回调的方式书写更简单，但是从ES6开始，我们有了async/await，这个语法改进跟Promise配合，能够有效地改善代码结构。

新特性：async/await

async/await是ES2016新加入的特性，它提供了用for、if等代码结构来编写异步的方式。它的运行时基础是Promise，面对这种比较新的特性，我们先来看一下基本用法。

async函数必定返回Promise，我们把所有返回Promise的函数都可以认为是异步函数。

async函数是一种特殊语法，特征是在function关键字之前加上async关键字，这样，就定义了一个async函数，我们可以在其中使用await来等待一个Promise。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(){
    console.log("a")
    await sleep(2000)
    console.log("b")
}
```

这段代码利用了我们之前定义的sleep函数。在异步函数foo中，我们调用sleep。

async函数强大之处在于，它是可以嵌套的。我们在定义了一批原子操作的情况下，可以利用async函数组合出新的async函数。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(name){
    await sleep(2000)
    console.log(name)
}
async function foo2(){
    await foo("a");
    await foo("b");
}
```

这里foo2用await调用了两次异步函数foo，可以看到，如果我们把sleep这样的异步操作放入某一个框架或者库中，使用者几乎不需要了解Promise的概念即可进行异步编程了。

此外，generator/iterator也常常被跟异步一起来讲，我们必须说明 generator/iterator 并非异步代码，只是在缺少async/await的时候，一些框架（最著名的要数co）使用这样的特性来模拟async/await。

但是generator并非被设计成实现异步，所以有了async/await之后，generator/iterator来模拟异步的方法应该被废弃。

结语

在今天的文章里，我们学习了JavaScript执行部分的知识，首先我们学习了JavaScript的宏观任务和微观任务相关的知识。我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任

务。许多的微观任务的队列组成了宏观任务。

除此之外，我们还展开介绍了用Promise来添加微观任务的方式，并且介绍了async/await这个语法的改进。

最后，留给你一个小练习：我们现在要实现一个红绿灯，把一个圆形div按照绿色3秒，黄色1秒，红色2秒循环改变背景色，你会怎样编写这个代码呢？欢迎你留言讨论。

你好，我是winter。这一部分我们来讲一讲JavaScript的执行。

首先我们考虑一下，如果我们是浏览器或者Node的开发者，我们该如何使用JavaScript引擎。

当拿到一段JavaScript代码时，浏览器或者Node环境首先要做的就是：传递给JavaScript引擎，并且要求它去执行。

然而，执行JavaScript并非一锤子买卖，宿主环境当遇到一些事件时，会继续把一段代码传递给JavaScript引擎去执行，此外，我们可能还会提供API给JavaScript引擎，比如setTimeout这样的API，它会允许JavaScript在特定的时机执行。

所以，我们首先应该形成一个感性的认知：一个JavaScript引擎会常驻于内存中，它等待着我们（宿主）把JavaScript代码或者函数传递给它执行。

在ES3和更早的版本中，JavaScript本身还没有异步执行代码的能力，这也就意味着，宿主环境传递给JavaScript引擎一段代码，引擎就把代码直接顺次执行了，这个任务也就是宿主发起的任务。

但是，在ES5之后，JavaScript引入了Promise，这样，不需要浏览器的安排，JavaScript引擎本身也可以发起任务了。

由于我们这里主要讲JavaScript语言，那么采纳JSC引擎的术语，我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任务。

宏观和微观任务

JavaScript引擎等待宿主环境分配宏观任务，在操作系统中，通常等待的行为都是一个事件循环，所以在Node术语中，也会把这个部分称为事件循环。

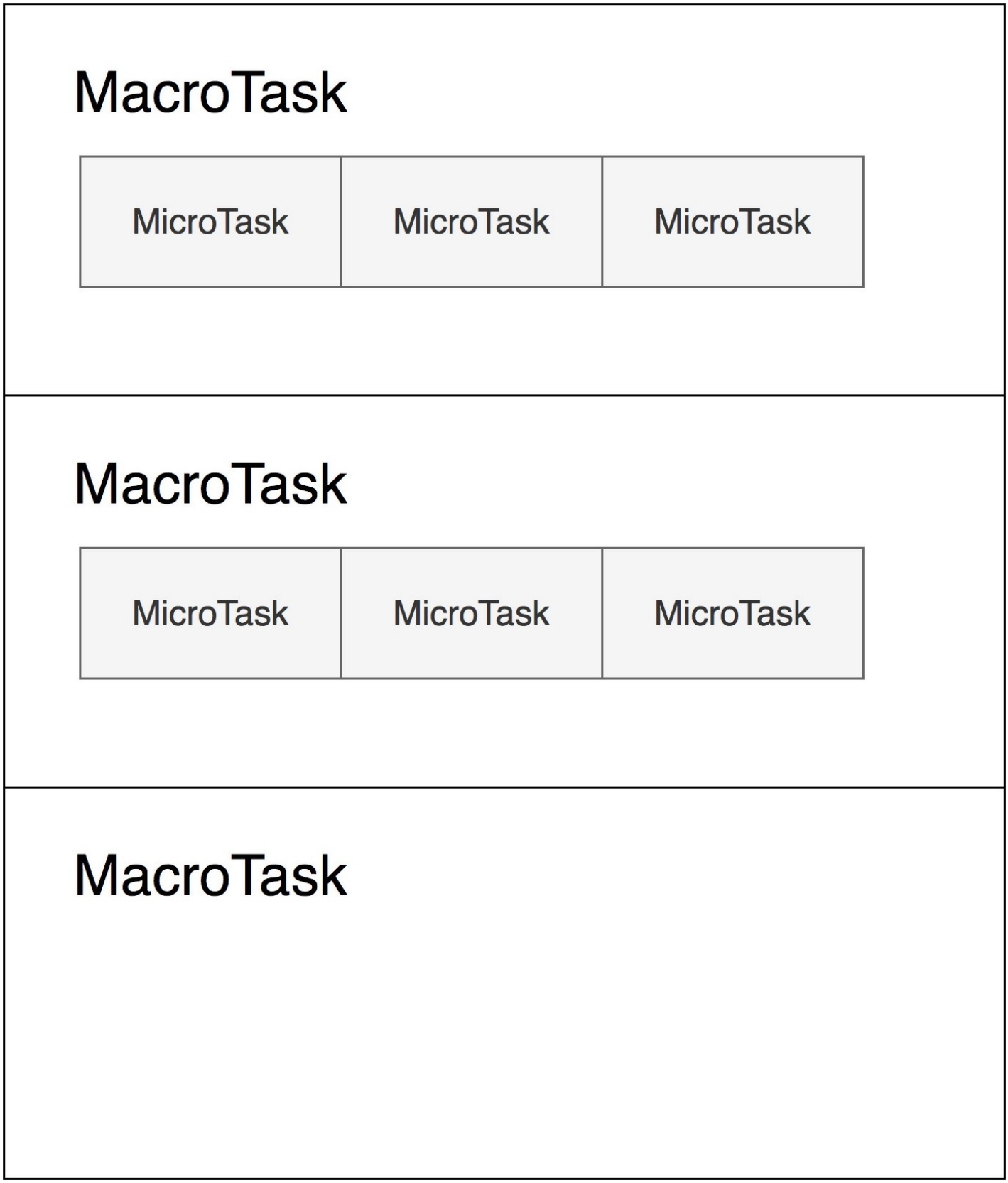
不过，术语本身并非我们需要重点讨论的内容，我们在这里把重点放在事件循环的原理上。在底层的C/C++代码中，这个事件循环是一个跑在独立线程中的循环，我们用伪代码来表示，大概是这样的：

```
while(TRUE) {
    r = wait();
    execute(r);
}
```

我们可以看到，整个循环做的事情基本上就是反复“等待-执行”。当然，实际的代码中并没有这么简单，还有要判断循环是否结束、宏观任务队列等逻辑，这里为了方便你理解，我就把这些都省略掉了。

这里每次的执行过程，其实都是一个宏观任务。我们可以大概理解：宏观任务的队列就相当于事件循环。

在宏观任务中，JavaScript的Promise还会产生异步代码，JavaScript必须保证这些异步代码在一个宏观任务中完成，因此，每个宏观任务中又包含了一个微观任务队列：



有了宏观任务和微观任务机制，我们就可以实现JavaScript引擎级和宿主级的任务了，例如：Promise永远在队列尾部添加微观任务。setTimeout等宿主API，则会添加宏观任务。

接下来，我们来详细介绍一下Promise。

Promise

Promise是JavaScript语言提供的一种标准化的异步管理方式，它的总体思想是，需要进行io、等待或者其它异步操作的函数，不返回真实结果，而返回一个“承诺”，函数的调用方可以在合适的时机，选择等待这个承诺兑现（通过Promise的then方法的回调）。

Promise的基本用法示例如下：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
sleep(1000).then( ()=> console.log("finished"));
```

这段代码定义了一个函数sleep，它的作用是等候传入参数指定的时长。

Promise的then回调是一个异步的执行过程，下面我们就来研究一下Promise函数中的执行顺序，我们来看一段代码示例：

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
r.then(() => console.log("c"));
console.log("b")
```

我们执行这段代码后，注意输出的顺序是 a b c。在进入console.log(“b”) 之前，毫无疑问 r 已经得到了resolve，但是Promise的resolve始终是异步操作，所以c无法出现在b之前。

接下来我们试试跟setTimeout混用的Promise。

在这段代码中，我设置了两段互不相干的异步操作：通过setTimeout执行console.log(“d”)，通过Promise执行console.log(“c”)。

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
setTimeout(()=>console.log("d"), 0)
r.then(() => console.log("c"));
console.log("b")
```

我们发现，不论代码顺序如何，d必定发生在c之后，因为Promise产生的是JavaScript引擎内部的微任务，而setTimeout是浏览器API，它产生宏任务。

为了理解微任务始终先于宏任务，我们设计一个实验：执行一个耗时1秒的Promise。

```
setTimeout(()=>console.log("d"), 0)
var r = new Promise(function(resolve, reject){
    resolve()
});
r.then(() => {
    var begin = Date.now();
    while(Date.now() - begin < 1000);
    console.log("c1")
    new Promise(function(resolve, reject){
        resolve()
    }).then(() => console.log("c2"))
});
```

这里我们强制了1秒的执行耗时，这样，我们可以确保任务c2是在d之后被添加到任务队列。

我们可以看到，即使耗时一秒的c1执行完毕，再enque的c2，仍然先于d执行了，这很好地解释了微任务优先的原理。

通过一系列的实验，我们可以总结一下如何分析异步执行的顺序：

- 首先我们分析有多少个宏任务；
- 在每个宏任务中，分析有多少个微任务；
- 根据调用次序，确定宏任务中的微任务执行次序；
- 根据宏任务的触发规则和调用次序，确定宏任务的执行次序；
- 确定整个顺序。

我们再来看一个稍微复杂的例子：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        console.log("b");
        setTimeout(resolve,duration);
    })
}
console.log("a");
sleep(5000).then(()=>console.log("c"));
```

这是一段非常常用的封装方法，利用Promise把setTimeout封装成可以用于异步的函数。

我们首先来看，setTimeout把整个代码分割成了2个宏观任务，这里不论是5秒还是0秒，都是一样的。

第一个宏观任务中，包含了先后同步执行的 console.log(“a”); 和 console.log(“b”);。

setTimeout后，第二个宏观任务执行调用了resolve，然后then中的代码异步得到执行，所以调用了console.log(“c”)，最终输出的顺序才是： a b c。

Promise是JavaScript中的一个定义，但是实际编写代码时，我们可以发现，它似乎并不比回调的方式书写更简单，但是从ES6开始，我们有了async/await，这个语法改进跟Promise配合，能够有效地改善代码结构。

新特性：async/await

async/await是ES2016新加入的特性，它提供了用for、if等代码结构来编写异步的方式。它的运行时基础是Promise，面对这种比较新的特性，我们先来看一下基本用法。

async函数必定返回Promise，我们把所有返回Promise的函数都可以认为是异步函数。

async函数是一种特殊语法，特征是在function关键字之前加上async关键字，这样，就定义了一个async函数，我们可以在其中使用await来等待一个Promise。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(){
    console.log("a")
    await sleep(2000)
    console.log("b")
}
```

这段代码利用了我们之前定义的sleep函数。在异步函数foo中，我们调用sleep。

async函数强大之处在于，它是可以嵌套的。我们在定义了一批原子操作的情况下，可以利用async函数组合出新的async函数。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(name){
    await sleep(2000)
    console.log(name)
}
async function foo2(){
    await foo("a");
    await foo("b");
}
```

这里foo2用await调用了两次异步函数foo，可以看到，如果我们把sleep这样的异步操作放入某一个框架或者库中，使用者几乎不需要了解Promise的概念即可进行异步编程了。

此外，generator/iterator也常常被跟异步一起来讲，我们必须说明 generator/iterator 并非异步代码，只是在缺少async/await的时候，一些框架（最著名的要数co）使用这样的特性来模拟async/await。

但是generator并非被设计成实现异步，所以有了async/await之后，generator/iterator来模拟异步的方法应该被废弃。

结语

在今天的文章里，我们学习了JavaScript执行部分的知识，首先我们学习了JavaScript的宏观任务和微观任务相关的知识。我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任

务。许多的微观任务的队列组成了宏观任务。

除此之外，我们还展开介绍了用Promise来添加微观任务的方式，并且介绍了async/await这个语法的改进。

最后，留给你一个小练习：我们现在要实现一个红绿灯，把一个圆形div按照绿色3秒，黄色1秒，红色2秒循环改变背景色，你会怎样编写这个代码呢？欢迎你留言讨论。

你好，我是winter。这一部分我们来讲一讲JavaScript的执行。

首先我们考虑一下，如果我们是浏览器或者Node的开发者，我们该如何使用JavaScript引擎。

当拿到一段JavaScript代码时，浏览器或者Node环境首先要做的就是：传递给JavaScript引擎，并且要求它去执行。

然而，执行JavaScript并非一锤子买卖，宿主环境当遇到一些事件时，会继续把一段代码传递给JavaScript引擎去执行，此外，我们可能还会提供API给JavaScript引擎，比如setTimeout这样的API，它会允许JavaScript在特定的时机执行。

所以，我们首先应该形成一个感性的认知：一个JavaScript引擎会常驻于内存中，它等待着我们（宿主）把JavaScript代码或者函数传递给它执行。

在ES3和更早的版本中，JavaScript本身还没有异步执行代码的能力，这也就意味着，宿主环境传递给JavaScript引擎一段代码，引擎就把代码直接顺次执行了，这个任务也就是宿主发起的任务。

但是，在ES5之后，JavaScript引入了Promise，这样，不需要浏览器的安排，JavaScript引擎本身也可以发起任务了。

由于我们这里主要讲JavaScript语言，那么采纳JSC引擎的术语，我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任务。

宏观和微观任务

JavaScript引擎等待宿主环境分配宏观任务，在操作系统中，通常等待的行为都是一个事件循环，所以在Node术语中，也会把这个部分称为事件循环。

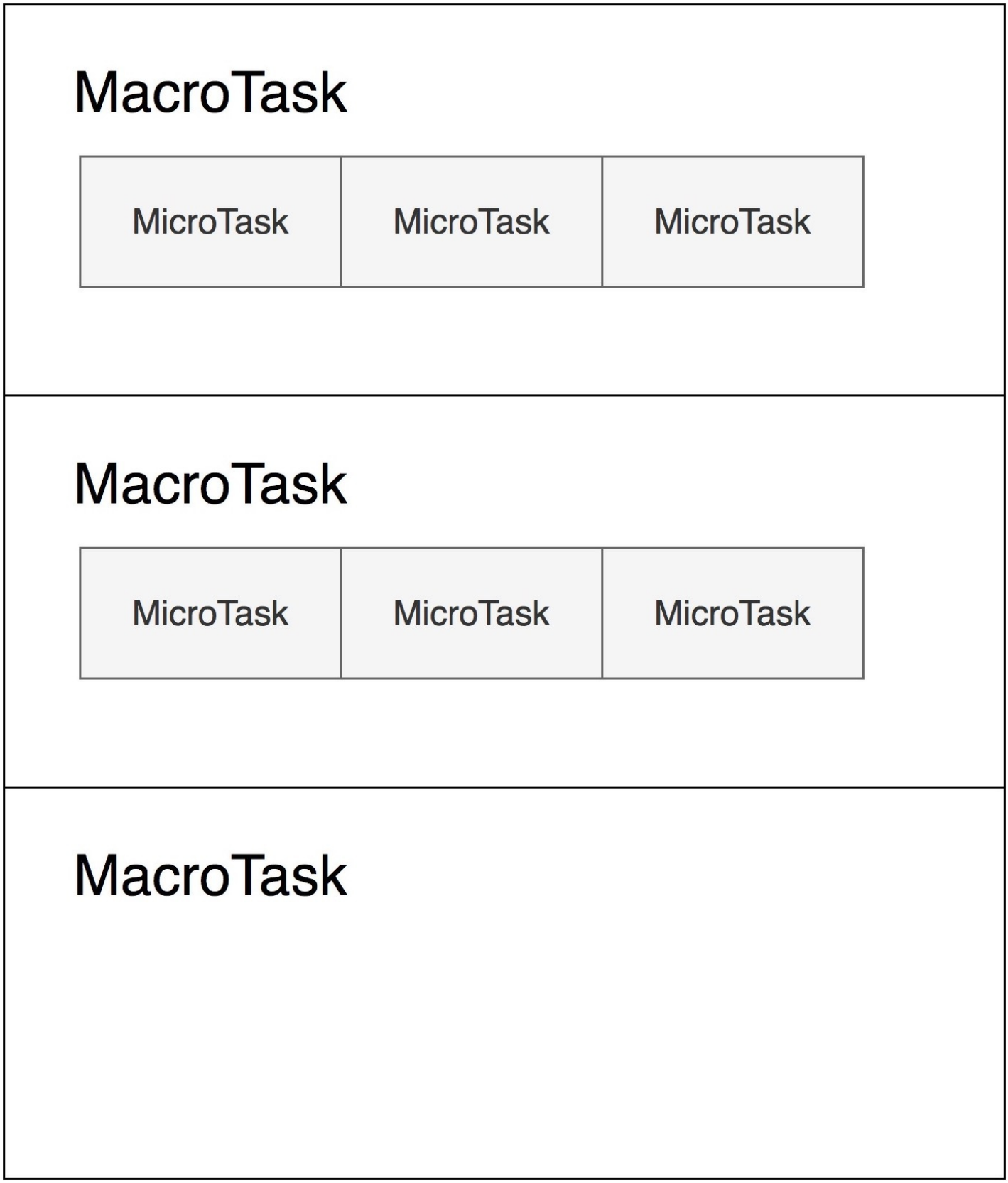
不过，术语本身并非我们需要重点讨论的内容，我们在这里把重点放在事件循环的原理上。在底层的C/C++代码中，这个事件循环是一个跑在独立线程中的循环，我们用伪代码来表示，大概是这样的：

```
while(TRUE) {
    r = wait();
    execute(r);
}
```

我们可以看到，整个循环做的事情基本上就是反复“等待-执行”。当然，实际的代码中并没有这么简单，还有要判断循环是否结束、宏观任务队列等逻辑，这里为了方便你理解，我就把这些都省略掉了。

这里每次的执行过程，其实都是一个宏观任务。我们可以大概理解：宏观任务的队列就相当于事件循环。

在宏观任务中，JavaScript的Promise还会产生异步代码，JavaScript必须保证这些异步代码在一个宏观任务中完成，因此，每个宏观任务中又包含了一个微观任务队列：



有了宏观任务和微观任务机制，我们就可以实现JavaScript引擎级和宿主级的任务了，例如：Promise永远在队列尾部添加微观任务。setTimeout等宿主API，则会添加宏观任务。

接下来，我们来详细介绍一下Promise。

Promise

Promise是JavaScript语言提供的一种标准化的异步管理方式，它的总体思想是，需要进行io、等待或者其它异步操作的函数，不返回真实结果，而返回一个“承诺”，函数的调用方可以在合适的时机，选择等待这个承诺兑现（通过Promise的then方法的回调）。

Promise的基本用法示例如下：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
sleep(1000).then( ()=> console.log("finished"));
```

这段代码定义了一个函数sleep，它的作用是等候传入参数指定的时长。

Promise的then回调是一个异步的执行过程，下面我们就来研究一下Promise函数中的执行顺序，我们来看一段代码示例：

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
r.then(() => console.log("c"));
console.log("b")
```

我们执行这段代码后，注意输出的顺序是 a b c。在进入console.log(“b”) 之前，毫无疑问 r 已经得到了resolve，但是Promise的resolve始终是异步操作，所以c无法出现在b之前。

接下来我们试试跟setTimeout混用的Promise。

在这段代码中，我设置了两段互不相干的异步操作：通过setTimeout执行console.log(“d”)，通过Promise执行console.log(“c”)。

```
var r = new Promise(function(resolve, reject){
    console.log("a");
    resolve()
});
setTimeout(()=>console.log("d"), 0)
r.then(() => console.log("c"));
console.log("b")
```

我们发现，不论代码顺序如何，d必定发生在c之后，因为Promise产生的是JavaScript引擎内部的微任务，而setTimeout是浏览器API，它产生宏任务。

为了理解微任务始终先于宏任务，我们设计一个实验：执行一个耗时1秒的Promise。

```
setTimeout(()=>console.log("d"), 0)
var r = new Promise(function(resolve, reject){
    resolve()
});
r.then(() => {
    var begin = Date.now();
    while(Date.now() - begin < 1000);
    console.log("c1")
    new Promise(function(resolve, reject){
        resolve()
    }).then(() => console.log("c2"))
});
```

这里我们强制了1秒的执行耗时，这样，我们可以确保任务c2是在d之后被添加到任务队列。

我们可以看到，即使耗时一秒的c1执行完毕，再enque的c2，仍然先于d执行了，这很好地解释了微任务优先的原理。

通过一系列的实验，我们可以总结一下如何分析异步执行的顺序：

- 首先我们分析有多少个宏任务；
- 在每个宏任务中，分析有多少个微任务；
- 根据调用次序，确定宏任务中的微任务执行次序；
- 根据宏任务的触发规则和调用次序，确定宏任务的执行次序；
- 确定整个顺序。

我们再来看一个稍微复杂的例子：

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        console.log("b");
        setTimeout(resolve,duration);
    })
}
console.log("a");
sleep(5000).then(()=>console.log("c"));
```

这是一段非常常用的封装方法，利用Promise把setTimeout封装成可以用于异步的函数。

我们首先来看，setTimeout把整个代码分割成了2个宏观任务，这里不论是5秒还是0秒，都是一样的。

第一个宏观任务中，包含了先后同步执行的 console.log(“a”); 和 console.log(“b”);。

setTimeout后，第二个宏观任务执行调用了resolve，然后then中的代码异步得到执行，所以调用了console.log(“c”)，最终输出的顺序才是： a b c。

Promise是JavaScript中的一个定义，但是实际编写代码时，我们可以发现，它似乎并不比回调的方式书写更简单，但是从ES6开始，我们有了async/await，这个语法改进跟Promise配合，能够有效地改善代码结构。

新特性：async/await

async/await是ES2016新加入的特性，它提供了用for、if等代码结构来编写异步的方式。它的运行时基础是Promise，面对这种比较新的特性，我们先来看一下基本用法。

async函数必定返回Promise，我们把所有返回Promise的函数都可以认为是异步函数。

async函数是一种特殊语法，特征是在function关键字之前加上async关键字，这样，就定义了一个async函数，我们可以在其中使用await来等待一个Promise。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(){
    console.log("a")
    await sleep(2000)
    console.log("b")
}
```

这段代码利用了我们之前定义的sleep函数。在异步函数foo中，我们调用sleep。

async函数强大之处在于，它是可以嵌套的。我们在定义了一批原子操作的情况下，可以利用async函数组合出新的async函数。

```
function sleep(duration) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve,duration);
    })
}
async function foo(name){
    await sleep(2000)
    console.log(name)
}
async function foo2(){
    await foo("a");
    await foo("b");
}
```

这里foo2用await调用了两次异步函数foo，可以看到，如果我们把sleep这样的异步操作放入某一个框架或者库中，使用者几乎不需要了解Promise的概念即可进行异步编程了。

此外，generator/iterator也常常被跟异步一起来讲，我们必须说明 generator/iterator 并非异步代码，只是在缺少async/await的时候，一些框架（最著名的要数co）使用这样的特性来模拟async/await。

但是generator并非被设计成实现异步，所以有了async/await之后，generator/iterator来模拟异步的方法应该被废弃。

结语

在今天的文章里，我们学习了JavaScript执行部分的知识，首先我们学习了JavaScript的宏观任务和微观任务相关的知识。我们把宿主发起的任务称为宏观任务，把JavaScript引擎发起的任务称为微观任

务。许多的微观任务的队列组成了宏观任务。

除此之外，我们还展开介绍了用Promise来添加微观任务的方式，并且介绍了async/await这个语法的改进。

最后，留给你一个小练习：我们现在要实现一个红绿灯，把一个圆形div按照绿色3秒，黄色1秒，红色2秒循环改变背景色，你会怎样编写这个代码呢？欢迎你留言讨论。