## **1.chunk**

* chunkGroup 由 chunk 组成，一个 chunkGroup 可以包含多个 chunk，在生成/优化 chunk graph 时会用到
* chunk 由 module 组成，一个 chunk 可以包含多个 module,它是 webpack 编译打包后输出的最终文件
* module 就是不同的资源文件，包含了你的代码中提供的例如：js/css/图片 等文件，在编译环节，webpack 会根据不同 module 之间的依赖关系去组合生成 chunk

## **2.seal**

* 根据 addEntry 方法中收集到入口文件组成的 \_preparedEntrypoints 数组

seal(callback) {

**this**.hooks.beforeChunks.call();

**for** (**const** preparedEntrypoint **of** **this**.\_preparedEntrypoints) {

**const** module = preparedEntrypoint.module;//入口模块

**const** name = preparedEntrypoint.name;//入口点名称 main

**const** chunk = **this**.addChunk(name);//新建chunk并添加到chunks中

**const** entrypoint = **new** Entrypoint(name);//生成入口点，其实是一个chunkGroup

entrypoint.setRuntimeChunk(chunk);//设置运行时chunk

entrypoint.addOrigin(null, name, preparedEntrypoint.request);//增加来源 ./src/index.js

**this**.namedChunkGroups.set(name, entrypoint);//key为chunk名称，值为chunkGroup

**this**.entrypoints.set(name, entrypoint);//入口点key为chunk名称，值为chunkGroup

**this**.chunkGroups.push(entrypoint);//添加一个新的chunkGroup

GraphHelpers.connectChunkGroupAndChunk(entrypoint, chunk);//建立起 chunkGroup 和 chunk 之间的关系

GraphHelpers.connectChunkAndModule(chunk, module);//建立起 chunk 和 module 之间的关系

chunk.entryModule = module;//代码块的入口模块

chunk.name = name;//代码块的名称

**this**.assignDepth(module);

}

}

## **3.buildChunkGraph**

* 遍历 module graph 模块依赖图建立起 basic chunk graph 依赖图
* 遍历第一步创建的 chunk graph 依赖图，依据之前的 module graph 来优化 chunk graph

### **3.1 文件**

#### **3.1.1 index.js**

**import** common **from** './common.js';**import**('./lazy.js').then(result => console.log(result))

#### **3.1.2 common.js**

**import** title **from** './title.js'

#### **3.1.3 lazy.js**

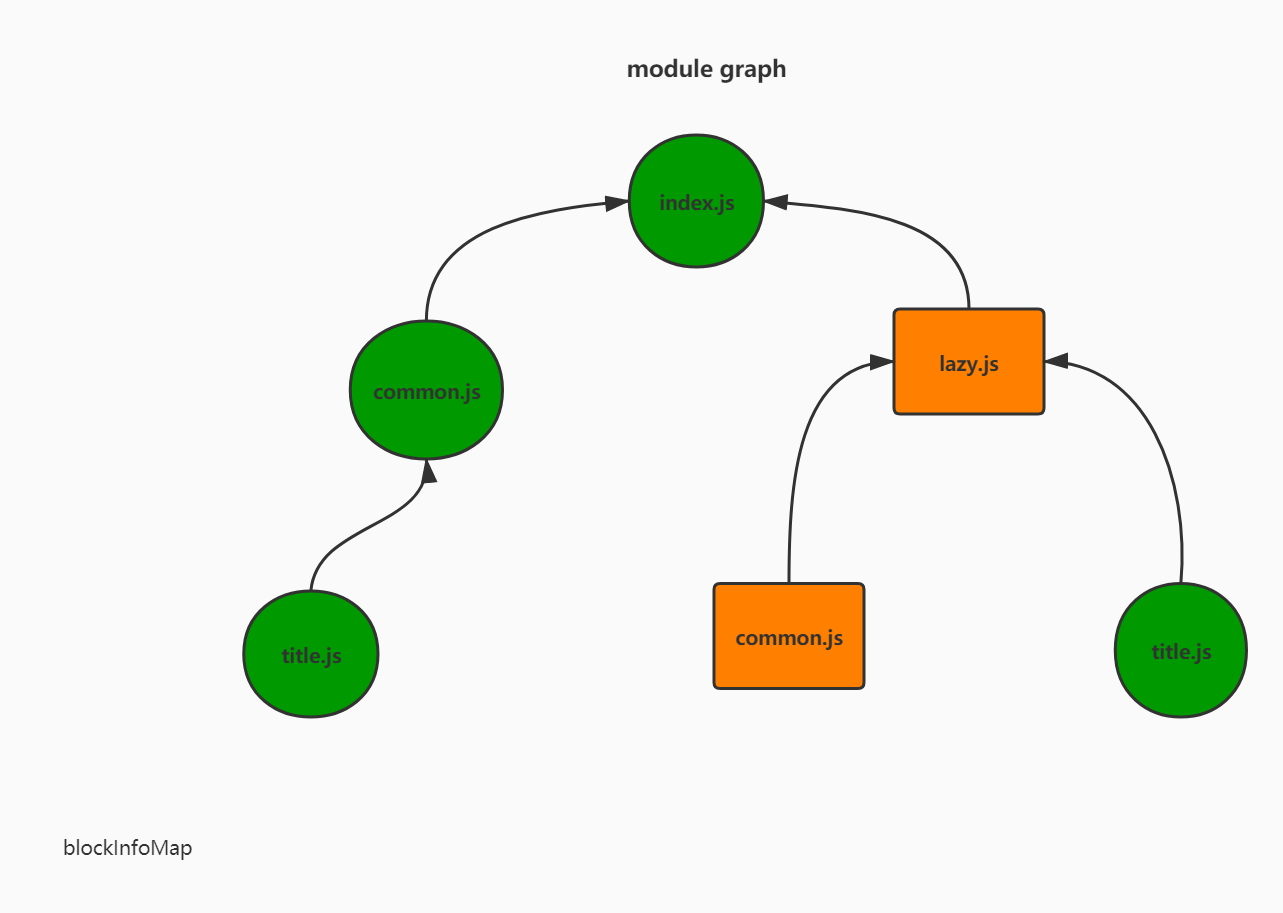
**import** title **from** './title.js';**import**('./common.js').then(result => console.log(result))**export** **const** lazy = 'lazy';

#### **3.1.4 title.js**

**export** **const** title = 'title';

### **3.2 module graph**

* 对这次 compilation 收集到的 modules 进行一次遍历
* 在遍历 module 的过程中，会对这个 module 的 dependencies 依赖进行处理
* 同时还会处理这个 module 的 blocks(即在你的代码通过异步 API 加载的模块),个异步 block 都会被加入到遍历的过程当中，被当做一个 module 来处理
* 遍历的过程结束后会建立起基本的 module graph，包含普通的 module 及异步 module(block)，最终存储到一个 map 表(blockInfoMap)当中
* [buildChunkGraph](https://github.com/webpack/webpack/blob/c9d4ff7b054fc581c96ce0e53432d44f9dd8ca72/lib/buildChunkGraph.js" \l "L138)



**const** extraceBlockInfoMap = compilation => {

**const** iteratorDependency = d => {

**const** ref = compilation.getDependencyReference(currentModule, d);

**if** (!ref) {

**return**;

}

//没有模块的跳过

**const** refModule = ref.module;

**if** (!refModule) {

**return**;

}

blockInfoModules.add(refModule);

};

**const** iteratorBlockPrepare = b => {

blockInfoBlocks.push(b);

blockQueue.push(b);//将 block 加入到 blockQueue 当中，从而进入到下一次的遍历过程当中

};

**let** currentModule;

**let** block;

**let** blockQueue;

**let** blockInfoModules;

**let** blockInfoBlocks;

**for** (**const** module **of** compilation.modules) {//循环所有的模块

blockQueue = [module];//基于此模块创建一个数组blockQueue

currentModule = module;//当前模块等于此模块

**while** (blockQueue.length > 0) {

block = blockQueue.pop();//取出一个模块

blockInfoModules = **new** Set();//唯一的模块集合

blockInfoBlocks = [];//块数组

**if** (block.variables) {//变量

**for** (**const** variable **of** block.variables) {

**for** (**const** dep **of** variable.dependencies) iteratorDependency(dep);

}

}

**if** (block.dependencies) {//普通依赖

**for** (**const** dep **of** block.dependencies) iteratorDependency(dep);

}

**if** (block.blocks) {//异步代码块依赖

**for** (**const** b **of** block.blocks) iteratorBlockPrepare(b);

}

**const** blockInfo = {

modules: blockInfoModules,

blocks: blockInfoBlocks

};

blockInfoMap.set(block, blockInfo);

}

}

**return** blockInfoMap;

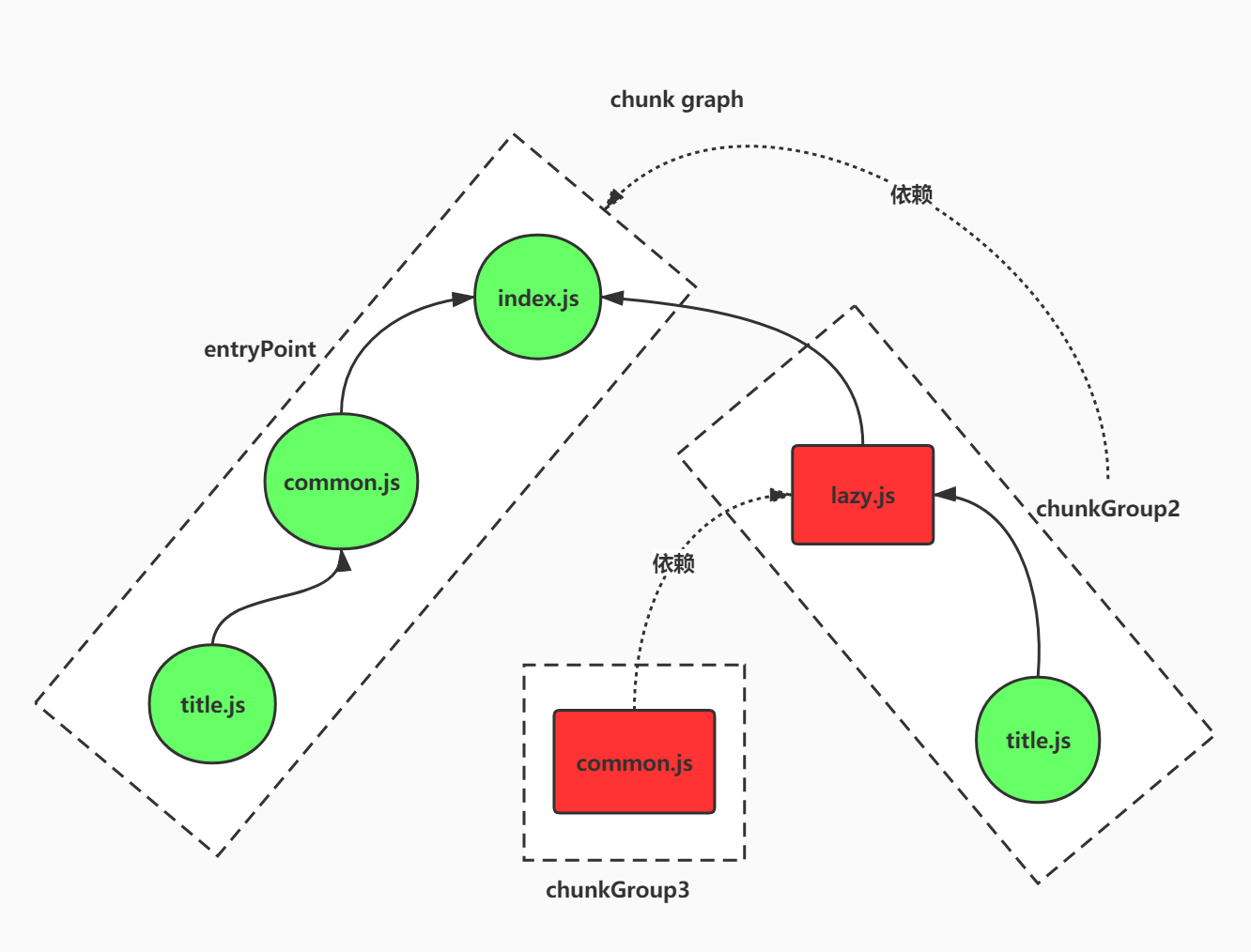
};

blockInfoMap

**class** **NormalModule** **extends** **Module** { } index.js**class** **ImportDependenciesBlock** **extends** **AsyncDependenciesBlock** { } lazy.js**class** **NormalModule** **extends** **Module** { } common.js**class** **NormalModule** **extends** **Module** { } title.js**class** **ImportDependenciesBlock** **extends** **AsyncDependenciesBlock** { } common.js**class** **NormalModule** **extends** **Module** { } title.js

### **3.3 生成chunk graph**

* [buildChunkGraph PART ONE](https://github.com/webpack/webpack/blob/c9d4ff7b054fc581c96ce0e53432d44f9dd8ca72/lib/buildChunkGraph.js" \l "L702)



#### **3.3.1 创建queue**

* 将传入的 entryPoint(chunkGroup) 转化为一个新的 queue
* chunkGroupInfoMap chunkGroup信息
* minAvailableModules 最小可跟踪的模块集合
* skippedItems 可以跳过的模块
* chunkGroupCounters key为chunkGroup,值为索引
* blockChunkGroups key为依赖块，值为chunkGroup
* allCreatedChunkGroups 所有创建的chunkGroup
* chunkDependencies key为chunkGroup,值为依赖的chunkGroup数组 {block,chunkGroup}
* queueConnect key为chunkGroup,值为一个依赖的chunkGroup数组
* availableModulesToBeMerged 父chunkGroup的模块
* outdatedChunkGroupInfo 过期的chunkGroup信息

**const** module = chunk.entryModule;

queue.push({

action: ENTER\_MODULE, //(需要被处理的模块类型，不同的处理类型的模块会经过不同的流程处理，初始为 ENTER\_MODULE(1))

block: module,//k (入口 module)

module,//(入口 module)

chunk,// (seal 阶段一开始为每个入口 module 创建的空 chunk)

chunkGroup//(entryPoint 即 chunkGroup 类型)

});

**const** visitModules = (

compilation,

inputChunkGroups,

chunkGroupInfoMap,

chunkDependencies,

blocksWithNestedBlocks,

allCreatedChunkGroups

) => {

**const** logger = compilation.getLogger("webpack.buildChunkGraph.visitModules");

**const** { namedChunkGroups } = compilation;

logger.time("prepare");

**const** blockInfoMap = extraceBlockInfoMap(compilation);

/\*\* @type {Map<ChunkGroup, { index: number, index2: number }>} \*/

**const** chunkGroupCounters = **new** Map();//计数器

**for** (**const** chunkGroup **of** inputChunkGroups) {

chunkGroupCounters.set(chunkGroup, {//每个chunkGroup有两个索引，默认值都是0

index: 0,

index2: 0

});

}

**let** nextFreeModuleIndex = 0;//下一个空闲的模块索引

**let** nextFreeModuleIndex2 = 0;//下一个空闲的模块索引

/\*\* @type {Map<DependenciesBlock, ChunkGroup>} \*/

**const** blockChunkGroups = **new** Map();

**const** ADD\_AND\_ENTER\_MODULE = 0;//增加并进入模块

**const** ENTER\_MODULE = 1;//进入模块

**const** PROCESS\_BLOCK = 2;//处理代码块

**const** LEAVE\_MODULE = 3;//离开模块

/\*\*

\* @param {QueueItem[]} queue the queue array (will be mutated)

\* @param {ChunkGroup} chunkGroup chunk group

\* @returns {QueueItem[]} the queue array again

\*/

**const** reduceChunkGroupToQueueItem = (queue, chunkGroup) => {

**for** (**const** chunk **of** chunkGroup.chunks) {

**const** module = chunk.entryModule;

queue.push({

action: ENTER\_MODULE,

block: module,

module,

chunk,

chunkGroup

});

}

chunkGroupInfoMap.set(chunkGroup, {

chunkGroup,

minAvailableModules: **new** Set(),

minAvailableModulesOwned: true,

availableModulesToBeMerged: [],

skippedItems: [],

resultingAvailableModules: undefined,

children: undefined

});

**return** queue;

};

// Start with the provided modules/chunks 把entryPoint(chunkGroup) 转化为一个新的 queue

/\*\* @type {QueueItem[]} \*/

**let** queue = inputChunkGroups

.reduce(reduceChunkGroupToQueueItem, [])

.reverse();

/\*\* @type {Map<ChunkGroup, Set<ChunkGroup>>} \*/

**const** queueConnect = **new** Map();

/\*\* @type {Set<ChunkGroupInfo>} \*/

**const** outdatedChunkGroupInfo = **new** Set();

/\*\* @type {QueueItem[]} \*/

**let** queueDelayed = [];

logger.timeEnd("prepare");

/\*\* @type {Module} \*/

**let** module;

/\*\* @type {Chunk} \*/

**let** chunk;

/\*\* @type {ChunkGroup} \*/

**let** chunkGroup;

/\*\* @type {DependenciesBlock} \*/

**let** block;

/\*\* @type {Set<Module>} \*/

**let** minAvailableModules;

/\*\* @type {QueueItem[]} \*/

**let** skippedItems;

// For each async Block in graph

/\*\*

\* @param {AsyncDependenciesBlock} b iterating over each Async DepBlock

\* @returns {void}

\*/

**const** iteratorBlock = b => {

// 1. We create a chunk for this Block

// but only once (blockChunkGroups map)

**let** c = blockChunkGroups.get(b);

**if** (c === undefined) {

c = namedChunkGroups.get(b.chunkName);

**if** (c && c.isInitial()) {

compilation.errors.push(

**new** AsyncDependencyToInitialChunkError(b.chunkName, module, b.loc)

);

c = chunkGroup;

} **else** {

//调用addChunkInGroup为这个异步的 block 新建一个 chunk 以及 chunkGroup

c = compilation.addChunkInGroup(

b.groupOptions || b.chunkName,

module,

b.loc,

b.request

);

//调用 GraphHelpers 模块提供的 connectChunkGroupAndChunk 建立起这个新建的 chunk 和 chunkGroup 之间的联系

//这里新建的 chunk 也就是在你的代码当中使用异步API 加载模块时，webpack 最终会单独给这个模块输出一个 chunk，但是此时这个 chunk 为一个空的 chunk，没有加入任何依赖的 module

chunkGroupCounters.set(c, { index: 0, index2: 0 });

blockChunkGroups.set(b, c);

allCreatedChunkGroups.add(c);

}

} **else** {

// TODO webpack 5 remove addOptions check

**if** (c.addOptions) c.addOptions(b.groupOptions);

c.addOrigin(module, b.loc, b.request);

}

// 2. We store the Block+Chunk mapping as dependency for the chunk

//建立起当前 module 所属的 chunkGroup 和 block 以及这个 block 所属的 chunkGroup 之间的依赖关系，并存储至 chunkDependencies Map 表中，

//这个 Map 表主要用于后面优化 chunk graph

**let** deps = chunkDependencies.get(chunkGroup);

**if** (!deps) chunkDependencies.set(chunkGroup, (deps = []));

deps.push({

block: b,

chunkGroup: c

});

// 3. We create/update the chunk group info

**let** connectList = queueConnect.get(chunkGroup);

**if** (connectList === undefined) {

connectList = **new** Set();

queueConnect.set(chunkGroup, connectList);

}

connectList.add(c);

// 4. We enqueue the DependenciesBlock for traversal

//向queueDelayed 中添加一个 action 类型为 PROCESS\_BLOCK,module 为当前所属的 module，block 为当前 module 依赖的异步模块

//chunk(chunkGroup 当中的第一个 chunk) 及 chunkGroup 都是处理异步模块生成的新项，而这里向 queueDelayed 数据集当中添加的新项主要就是用于 queue 的外层遍历

queueDelayed.push({

action: PROCESS\_BLOCK,

block: b,

module: module,

chunk: c.chunks[0],

chunkGroup: c

});

};

// Iterative traversal of the Module graph

// Recursive would be simpler to write but could result in Stack Overflows

//开始进入到外层的遍历当中，即对 queueDelayed 数据集进行处理

**while** (queue.length) {

logger.time("visiting");

//每一轮的内层遍历都对应于同一个 chunkGroup，即每一轮内层的遍历都是对这个 chunkGroup 当中所包含的所有的 module 进行处理

**while** (queue.length) {

**const** queueItem = queue.pop();

module = queueItem.module;

block = queueItem.block;

chunk = queueItem.chunk;

**if** (chunkGroup !== queueItem.chunkGroup) {

chunkGroup = queueItem.chunkGroup;

**const** chunkGroupInfo = chunkGroupInfoMap.get(chunkGroup);

minAvailableModules = chunkGroupInfo.minAvailableModules;

skippedItems = chunkGroupInfo.skippedItems;

}

**switch** (queueItem.action) {

**case** ADD\_AND\_ENTER\_MODULE: {

**if** (minAvailableModules.has(module)) {

// already in parent chunks

// skip it for now, but enqueue for rechecking when minAvailableModules shrinks

skippedItems.push(queueItem);

**break**;

}

// We connect Module and Chunk when not already done

**if** (chunk.addModule(module)) {

module.addChunk(chunk);

} **else** {

// already connected, skip it

**break**;

}

}

// fallthrough

//在 queue 中新增一个 action 为 LEAVE\_MODULE 的项会在后面遍历的流程当中使用

**case** ENTER\_MODULE: {

**if** (chunkGroup !== undefined) {

**const** index = chunkGroup.getModuleIndex(module);

**if** (index === undefined) {

chunkGroup.setModuleIndex(

module,

chunkGroupCounters.get(chunkGroup).index++

);

}

}

**if** (module.index === null) {

module.index = nextFreeModuleIndex++;

}

queue.push({

action: LEAVE\_MODULE,

block,

module,

chunk,

chunkGroup

});

}

// fallthrough

//当 ENTRY\_MODULE 的阶段进行完后，立即进入到了 PROCESS\_BLOCK 阶段

**case** PROCESS\_BLOCK: {

// 根据 module graph 依赖图保存的模块映射 blockInfoMap 获取这个 module（称为A） 的同步依赖 modules 及异步依赖 blocks

**const** blockInfo = blockInfoMap.get(block);

// Buffer items because order need to be reverse to get indicies correct

**const** skipBuffer = [];

**const** queueBuffer = [];

// Traverse all referenced modules

**for** (**const** refModule **of** blockInfo.modules) {

/\*\*

\* 判断当前这个 module(A) 所属的 chunk 当中是否包含了其依赖 modules 当中的 module(B)，

\* 如果不包含的话，那么会在 queue 当中加入新的项，新加入的项目的 action 为 ADD\_AND\_ENTER\_MODULE，

\* 即这个新增项在下次遍历的时候，首先会进入到 ADD\_AND\_ENTER\_MODULE 阶段

\*/

**if** (chunk.containsModule(refModule)) {

// skip early if already connected

**continue**;

}

**if** (minAvailableModules.has(refModule)) {

// already in parent chunks, skip it for now

skipBuffer.push({

action: ADD\_AND\_ENTER\_MODULE,

block: refModule,

module: refModule,

chunk,

chunkGroup

});

**continue**;

}

// enqueue the add and enter to enter in the correct order

// this is relevant with circular dependencies

queueBuffer.push({

action: ADD\_AND\_ENTER\_MODULE,

block: refModule,

module: refModule,

chunk,

chunkGroup

});

}

// Add buffered items in reversed order

**for** (**let** i = skipBuffer.length - 1; i >= 0; i--) {

skippedItems.push(skipBuffer[i]);

}

**for** (**let** i = queueBuffer.length - 1; i >= 0; i--) {

queue.push(queueBuffer[i]);

}

//调用iteratorBlock方法来处理这个 module(A) 依赖的所有的异步 blocks

**for** (**const** block **of** blockInfo.blocks) iteratorBlock(block);

**if** (blockInfo.blocks.length > 0 && module !== block) {

blocksWithNestedBlocks.add(block);

}

**break**;

}

**case** LEAVE\_MODULE: {

**if** (chunkGroup !== undefined) {

**const** index = chunkGroup.getModuleIndex2(module);

**if** (index === undefined) {

chunkGroup.setModuleIndex2(

module,

chunkGroupCounters.get(chunkGroup).index2++

);

}

}

**if** (module.index2 === null) {

module.index2 = nextFreeModuleIndex2++;

}

**break**;

}

}

}

logger.timeEnd("visiting");

**while** (queueConnect.size > 0) {

logger.time("calculating available modules");

//计算出这个chunkGroup新的父模块，以便这些子chunkGroup可以获取新的模块

// Figure out new parents for chunk groups

// to get new available modules for these children

**for** (**const** [chunkGroup, targets] **of** queueConnect) {

**const** info = chunkGroupInfoMap.get(chunkGroup);//当前信息

**let** minAvailableModules = info.minAvailableModules;//(chunkGroup 可追踪的最小 module 数据集)

//为这个点创建一个新的模块数据集,添加所有的本chunkgroup所有的chunk里的模块

**const** resultingAvailableModules = **new** Set(minAvailableModules);

**for** (**const** chunk **of** chunkGroup.chunks) {

**for** (**const** m **of** chunk.modulesIterable) {

resultingAvailableModules.add(m);

}

}

info.resultingAvailableModules = resultingAvailableModules;

**if** (info.children === undefined) {

info.children = targets;

} **else** {

**for** (**const** target **of** targets) {

info.children.add(target);

}

}

// 2. Update chunk group info 更新chunkGroup信息

**for** (**const** target **of** targets) {

**let** chunkGroupInfo = chunkGroupInfoMap.get(target);

**if** (chunkGroupInfo === undefined) {

chunkGroupInfo = {

chunkGroup: target,

minAvailableModules: undefined,

minAvailableModulesOwned: undefined,

availableModulesToBeMerged: [],

skippedItems: [],

resultingAvailableModules: undefined,

children: undefined

};

chunkGroupInfoMap.set(target, chunkGroupInfo);

}

chunkGroupInfo.availableModulesToBeMerged.push(

resultingAvailableModules

);

outdatedChunkGroupInfo.add(chunkGroupInfo);

}

}

queueConnect.clear();

logger.timeEnd("calculating available modules");

**if** (outdatedChunkGroupInfo.size > 0) {

logger.time("merging available modules");//合并模块

// Execute the merge

**for** (**const** info **of** outdatedChunkGroupInfo) {

**const** availableModulesToBeMerged = info.availableModulesToBeMerged;//父模块

**let** cachedMinAvailableModules = info.minAvailableModules;//最小化模块集

// 1. Get minimal available modules

// It doesn't make sense to traverse a chunk again with more available modules.

// This step calculates the minimal available modules and skips traversal when

// the list didn't shrink.

**if** (availableModulesToBeMerged.length > 1) {

availableModulesToBeMerged.sort(bySetSize);

}

**let** changed = false;

**for** (**const** availableModules **of** availableModulesToBeMerged) {

**if** (cachedMinAvailableModules === undefined) {

cachedMinAvailableModules = availableModules;

info.minAvailableModules = cachedMinAvailableModules;

info.minAvailableModulesOwned = false;

changed = true;

} **else** {

**if** (info.minAvailableModulesOwned) {

// We own it and can modify it

**for** (**const** m **of** cachedMinAvailableModules) {

**if** (!availableModules.has(m)) {

cachedMinAvailableModules.delete(m);

changed = true;

}

}

} **else** {

**for** (**const** m **of** cachedMinAvailableModules) {

**if** (!availableModules.has(m)) {

// cachedMinAvailableModules need to be modified

// but we don't own it

// construct a new Set as intersection of cachedMinAvailableModules and availableModules

/\*\* @type **{Set<Module>}** \*/

**const** newSet = **new** Set();

**const** iterator = cachedMinAvailableModules[

Symbol.iterator

]();

/\*\* @type {IteratorResult<Module>} \*/

**let** it;

**while** (!(it = iterator.next()).done) {

**const** module = it.value;

**if** (module === m) **break**;

newSet.add(module);

}

**while** (!(it = iterator.next()).done) {

**const** module = it.value;

**if** (availableModules.has(module)) {

newSet.add(module);

}

}

cachedMinAvailableModules = newSet;

info.minAvailableModulesOwned = true;

info.minAvailableModules = newSet;

// Update the cache from the first queue

// if the chunkGroup is currently cached

**if** (chunkGroup === info.chunkGroup) {

minAvailableModules = cachedMinAvailableModules;

}

changed = true;

**break**;

}

}

}

}

}

availableModulesToBeMerged.length = 0;

**if** (!changed) **continue**;

// 2. Reconsider skipped items

**for** (**const** queueItem **of** info.skippedItems) {

queue.push(queueItem);

}

info.skippedItems.length = 0;

// 3. Reconsider children chunk groups

**if** (info.children !== undefined) {

**const** chunkGroup = info.chunkGroup;

**for** (**const** c **of** info.children) {

**let** connectList = queueConnect.get(chunkGroup);

**if** (connectList === undefined) {

connectList = **new** Set();

queueConnect.set(chunkGroup, connectList);

}

connectList.add(c);

}

}

}

outdatedChunkGroupInfo.clear();

logger.timeEnd("merging available modules");

}

}

//当队列中所有的元素处理完成后处理queueDelayed

//这个对获取合局正确的索引非常重要

//异步blocks应该在所有的同步blocks处理完成后再处理

**if** (queue.length === 0) {

**const** tempQueue = queue;

queue = queueDelayed.reverse();

queueDelayed = tempQueue;

}

}

};

### **3.4 优化chunk graph**

* [buildChunkGraph PART TWO](https://github.com/webpack/webpack/blob/c9d4ff7b054fc581c96ce0e53432d44f9dd8ca72/lib/buildChunkGraph.js" \l "L713)
* [afterChunks](https://github.com/webpack/webpack/blob/c9d4ff7b054fc581c96ce0e53432d44f9dd8ca72/lib/Compilation.js" \l "L1320)

/\*\*

\*

\* @param **{Set<DependenciesBlock>}** blocksWithNestedBlocks block拥有子block的标识

\* @param **{Map<ChunkGroup, ChunkGroupDep[]>}** chunkDependencies chunk groups的依赖

\* @param **{Map<ChunkGroup, ChunkGroupInfo>}** chunkGroupInfoMap chunkgroup到可用的modules的映射

\*/**const** connectChunkGroups = (

blocksWithNestedBlocks,

chunkDependencies,

chunkGroupInfoMap

) => {

/\*\* @type **{Set<Module>}** \*/

**let** resultingAvailableModules;

/\*\*

\* 帮助函数，检查是否代码块中是否已经有了所有模块

\* @param {ChunkGroup} chunkGroup 要检查的chunkGroup

\* @param {Set<Module>} availableModules 模块集合

\* @returns {boolean} 如果所有的模块都在这个chunk中存在就返回true

\*/

**const** areModulesAvailable = (chunkGroup, availableModules) => {

**for** (**const** chunk **of** chunkGroup.chunks) {

**for** (**const** module **of** chunk.modulesIterable) {

**if** (!availableModules.has(module)) **return** false;

}

}

**return** true;

};

// For each edge in the basic chunk graph

/\*\*

\* @param {ChunkGroupDep} dep the dependency used for filtering

\* @returns {boolean} used to filter "edges" (aka Dependencies) that were pointing

\* to modules that are already available. Also filters circular dependencies in the chunks graph

\*/

**const** filterFn = dep => {

**const** depChunkGroup = dep.chunkGroup;

// TODO is this needed?

**if** (blocksWithNestedBlocks.has(dep.block)) **return** true;

**if** (areModulesAvailable(depChunkGroup, resultingAvailableModules)) {

**return** false; // break all modules are already available

}

**return** true;

};

//遍历所有的deps,检查是否这个chunkGroup是否需要连接

**for** (**const** [chunkGroup, deps] **of** chunkDependencies) {

**if** (deps.length === 0) **continue**;

// 1.从map中获取到chunkGroup的信息对象

**const** info = chunkGroupInfoMap.get(chunkGroup);

resultingAvailableModules = info.resultingAvailableModules;

// 2. Foreach edge

**for** (**let** i = 0; i < deps.length; i++) {

**const** dep = deps[i];

// Filter inline, rather than creating a new array from `.filter()`

// TODO check if inlining filterFn makes sense here

//判断 chunkGroup 提供的 newAvailableModules(可以将 newAvailableModules 理解为这个 chunkGroup 所有 module 的集合setA)和

// deps 依赖中的 chunkGroup (由异步 block 创建的 chunkGroup)所包含的 chunk 当中所有的 module 集合(setB)包含关系：

**if** (!filterFn(dep)) {

**continue**;

}

**const** depChunkGroup = dep.chunkGroup;

**const** depBlock = dep.block;

// 5. 建立起 deps 依赖中的异步 block 和 chunkGroup 的依赖关系

GraphHelpers.connectDependenciesBlockAndChunkGroup(

depBlock,

depChunkGroup

);

// 6. chunkGroup 和 deps 依赖中的 chunkGroup 之间的依赖关系

//（这个依赖关系也决定了在 webpack 编译完成后输出的文件当中是否会有 deps 依赖中的 chunkGroup 所包含的 chunk）

GraphHelpers.connectChunkGroupParentAndChild(chunkGroup, depChunkGroup);

}

}

};

/\*\*

\* Remove all unconnected chunk groups

\* @param {Compilation} compilation the compilation

\* @param {Iterable<ChunkGroup>} allCreatedChunkGroups all chunk groups that where created before

\*/**const** cleanupUnconnectedGroups = (compilation, allCreatedChunkGroups) => {

**for** (**const** chunkGroup **of** allCreatedChunkGroups) {

**if** (chunkGroup.getNumberOfParents() === 0) {//开始处理没有依赖关系的 chunkGroup

**for** (**const** chunk **of** chunkGroup.chunks) {

**const** idx = compilation.chunks.indexOf(chunk);

**if** (idx >= 0) compilation.chunks.splice(idx, 1);

chunk.remove("unconnected");

}

chunkGroup.remove("unconnected");

}

}

};

