link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=116 sentences=267, words=2047

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;
            const name = preparedEntrypoint.name;
            const chunk = this.addChunk(name);
            const entrypoint = new Entrypoint(name);
            entrypoint.setRuntimeChunk(chunk);
            entrypoint.addOrigin(null, name, preparedEntrypoint.request);
            this.namedChunkGroups.set(name, entrypoint);
            this.entrypoints.set(name, entrypoint);
            this.chunkGroups.push(entrypoint);
            GraphHelpers.connectChunkGroupAndChunk(entrypoint, chunk);
            GraphHelpers.connectChunkAndModule(chunk, module);
            chunk.entryModule = module;
            chunk.name = name;
            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

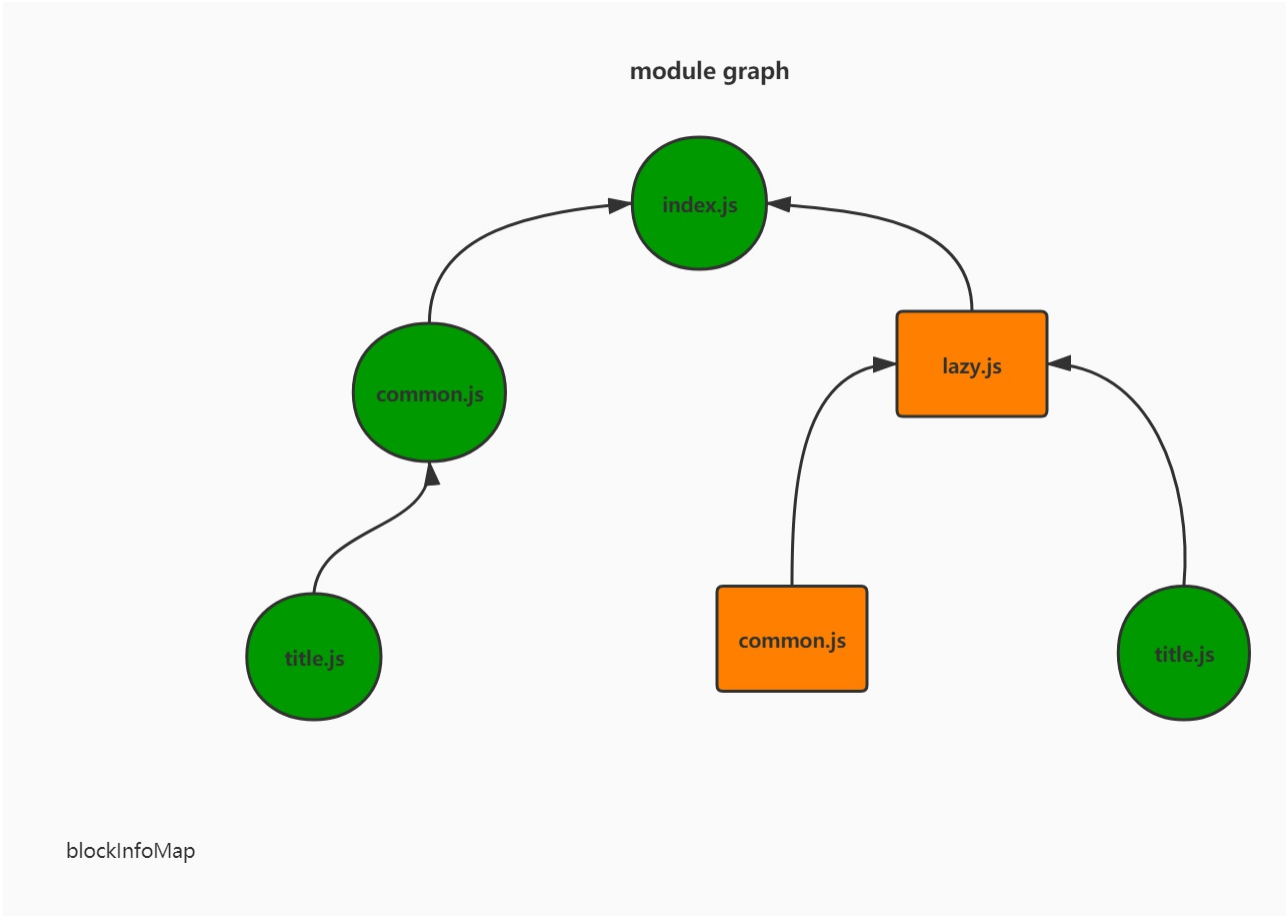
```
import title from './title.js';
import('./common.js').then(result => console.log(result))
export const lazy = 'lazy';
```

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#L138)



blockInfoMap

```
const extraceBlockInfoMap = compilation => {
    const iteratorDependency = d => {
        const ref = compilation.getDependencyReference(currentModule, d);
        if (!ref) {
            return;
        }

        const refModule = ref.module;
        if (!refModule) {
            return;
        }
        blockInfoModules.add(refModule);
    };

    const iteratorBlockPrepare = b => {
        blockInfoBlocks.push(b);
        blockQueue.push(b);
    };

    let currentModule;
    let block;
    let blockQueue;
    let blockInfoModules;
    let blockInfoBlocks;

    for (const module of compilation.modules) {
        blockQueue = [module];
        currentModule = module;
        while (blockQueue.length > 0) {
            block = blockQueue.pop();
            blockInfoModules = new Set();
            blockInfoBlocks = [];

            if (block.variables) {
                for (const variable of block.variables) {
                    for (const dep of variable.dependencies) iteratorDependency(dep);
                }
            }

            if (block.dependencies) {
                for (const dep of block.dependencies) iteratorDependency(dep);
            }

            if (block.blocks) {
                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#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,
    block: module,
    module,
    chunk,
    chunkGroup
});
```

```
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);

const chunkGroupCounters = new Map();
for (const chunkGroup of inputChunkGroups) {
    chunkGroupCounters.set(chunkGroup, {
        index: 0,
        index2: 0
    });
}

let nextFreeModuleIndex = 0;
let nextFreeModuleIndex2 = 0;

const blockChunkGroups = new Map();

const ADD_AND_ENTER_MODULE = 0;
const ENTER_MODULE = 1;
const PROCESS_BLOCK = 2;
const LEAVE_MODULE = 3;

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;
};

let queue = inputChunkGroups
    .reduce(reduceChunkGroupToQueueItem, [])
    .reverse();

const queueConnect = new Map();

const outdatedChunkGroupInfo = new Set();

let queueDelayed = [];

logger.timeEnd("prepare");

let module;

let chunk;

let chunkGroup;

let block;

let minAvailableModules;

let skippedItems;

const iteratorBlock = b => {

    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 {

            c = compilation.addChunkInGroup(
                b.groupOptions || b.chunkName,
                module,
                b.loc,
                b.request
            );

            chunkGroupCounters.set(c, { index: 0, index2: 0 });
            blockChunkGroups.set(b, c);
            allCreatedChunkGroups.add(c);
        }
    } else {

        if (c.addOptions) c.addOptions(b.groupOptions);
        c.addOrigin(module, b.loc, b.request);
    }

    let deps = chunkDependencies.get(chunkGroup);
    if (!deps) chunkDependencies.set(chunkGroup, (deps = []));
    deps.push({
        block: b,
        chunkGroup: c
```

```
    });

    let connectList = queueConnect.get(chunkGroup);
    if (connectList === undefined) {
        connectList = new Set();
        queueConnect.set(chunkGroup, connectList);
    }
    connectList.add(c);

    queueDelayed.push({
        action: PROCESS_BLOCK,
        block: b,
        module: module,
        chunk: c.chunks[0],
        chunkGroup: c
    });
};

while (queue.length) {
    logger.time("visiting");

    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)) {

                    skippedItems.push(queueItem);
                    break;
                }

                if (chunk.addModule(module)) {
                    module.addChunk(chunk);
                } else {

                    break;
                }
            }

            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
                });
            }

            case PROCESS_BLOCK: {

                const blockInfo = blockInfoMap.get(block);

                const skipBuffer = [];
                const queueBuffer = [];

                for (const refModule of blockInfo.modules) {

                    if (chunk.containsModule(refModule)) {

                        continue;
                    }
                    if (minAvailableModules.has(refModule)) {

                        skipBuffer.push({
                            action: ADD_AND_ENTER_MODULE,
                            block: refModule,
                            module: refModule,
                            chunk,
                            chunkGroup
                        });
                        continue;
                    }

                    queueBuffer.push({
                        action: ADD_AND_ENTER_MODULE,
                        block: refModule,
                        module: refModule,
                        chunk,
                        chunkGroup
```

```
                });
            }

            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]);
            }

            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");

    for (const [chunkGroup, targets] of queueConnect) {
        const info = chunkGroupInfoMap.get(chunkGroup);
        let minAvailableModules = info.minAvailableModules;

        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);
            }
        }

        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");

        for (const info of outdatedChunkGroupInfo) {
            const availableModulesToBeMerged = info.availableModulesToBeMerged;
            let cachedMinAvailableModules = info.minAvailableModules;

            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) {

                        for (const m of cachedMinAvailableModules) {
                            if (!availableModules.has(m)) {
```

```
                                    cachedMinAvailableModules.delete(m);
                                    changed = true;
                                }
                            }
                        } else {
                            for (const m of cachedMinAvailableModules) {
                                if (!availableModules.has(m)) {

                                    const newSet = new Set();
                                    const iterator = cachedMinAvailableModules[
                                        Symbol.iterator
                                    ]();

                                    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;

                                    if (chunkGroup === info.chunkGroup) {
                                        minAvailableModules = cachedMinAvailableModules;
                                    }

                                    changed = true;
                                    break;
                                }
                            }
                        }
                    }
                }
                availableModulesToBeMerged.length = 0;
                if (!changed) continue;

                for (const queueItem of info.skippedItems) {
                    queue.push(queueItem);
                }
                info.skippedItems.length = 0;

                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");
        }
    }

    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#L713)
- afterChunks (https://github.com/webpack/webpack/blob/c9d4ff7b054fc581c96ce0e53432d44f9dd8ca72/lib/Compilation.js#L1320)

```
const connectChunkGroups = (
    blocksWithNestedBlocks,
    chunkDependencies,
    chunkGroupInfoMap
) => {

    let resultingAvailableModules;

    const areModulesAvailable = (chunkGroup, availableModules) => {
        for (const chunk of chunkGroup.chunks) {
            for (const module of chunk.modulesIterable) {
                if (!availableModules.has(module)) return false;
            }
        }
        return true;
    };

    const filterFn = dep => {
        const depChunkGroup = dep.chunkGroup;

        if (blocksWithNestedBlocks.has(dep.block)) return true;
        if (areModulesAvailable(depChunkGroup, resultingAvailableModules)) {
            return false;
        }
        return true;
    };

    for (const [chunkGroup, deps] of chunkDependencies) {
        if (deps.length === 0) continue;

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

        for (let i = 0; i < deps.length; i++) {
            const dep = deps[i];

            if (!filterFn(dep)) {
                continue;
            }
            const depChunkGroup = dep.chunkGroup;
            const depBlock = dep.block;

            GraphHelpers.connectDependenciesBlockAndChunkGroup(
                depBlock,
                depChunkGroup
            );

            GraphHelpers.connectChunkGroupParentAndChild(chunkGroup, depChunkGroup);
        }
    }
};

const cleanupUnconnectedGroups = (compilation, allCreatedChunkGroups) => {
    for (const chunkGroup of allCreatedChunkGroups) {
        if (chunkGroup.getNumberOfParents() === 0) {
            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");
        }
    }
};
```