

superbitch bot 开发笔记

前言

该笔记是由 **screeps** 游戏玩家 **superbitch** 书写的基于他的 **bot** 框架的说明性文档。文档中会详细说明每个模块的作用、用法甚至实例，以便读者可以根据自己需要个性化的添加任务、或修改参数。书写过程中遇到冗余的设计也会强调，他人可以根据自己需要修改。然后由于代码格式不规范或可能不符合他人习惯，可以事先使用格式修改插件对该框架代码进行格式优化后再观看。

项目地址

<https://gitee.com/mikebraton/xscreeps> (仍在开发中)

写该文档时的版本分支: e3f0e98d854cb90dd781ca5f3431ac29e88b5310

[建议在该版本下进行开发]

本项目的优势与不足

目前看来，我的项目存在一些优势和不足。**优势**：部分模块化的设计、可以根据自己的需要自定义任务、项目中不存在任何难以理解的代码及函数【最大优势】、自带主动防御、资源调度、四人小队等模块任务。**不足**：过渡设计导致的冗余严重(有很多完全没用到的类型、函数等出现在了项目中)、cpu 优化相对其他大佬还是差一截(大概不挖过道、不开外矿、不考虑额外打架 cpu，s3 能支持 8~10 个房间|挖过道 7-8 个房间)、基于原型拓展的框架(可能有些人不喜欢这样的方式)

如何开发本项目

使用 vscode 开发的准备工作

请详细阅读 hoho 的项目的 readme.md 【master】

<https://github.com/HoPGoldy/my-screeps-ai>

作者

本项目由 **superbitch** 开发，**E19N2**（人名）也作为开发人员开发了部分功能。作为一个机械行业的工科生，并不擅长编程，所以写的代码槽点很多，水平也不足。如果有好的建议，望不吝赐教。

各文件作用说明 [注:所有 xxx.interface.ts 文件均为类型定义文件 不说明]

src/

- _errorMap/ : 报错信息映射相关
- boot/ : 存放房间运行主文件和爬虫运行主文件
 - creepWork.ts 爬虫运行主文件 (包含 pc 和普通爬 也涉及跨 shard 的部分逻辑)
 - roomWork.ts 房间运行主文件
- constant/ 存放常量信息
 - BoostConstant.ts boost 时用到的常量
 - PlanConstant.ts 房间布局常量
 - ResourceConstant.ts 资源常量、合成化合物相关常量
 - SpawnConstant.ts 孵化常量 各角色爬虫的信息都在里面
- interface/
- module/ 存放模块文件
 - dispatch/ 资源调度模块
 - fun/ 较杂的不好定义的函数模块都放里面
 - global/ 全局相关模块
 - layoutVisual/ 一个布局可视化的模块
 - shard/ 跨 shard 模块
 - squad/ 四人小队模块
- mount/ 原型拓展相关 【重要】
 - console/ 控制台 api 相关拓展
 - control/ 常规 api 例如: 攻击、采矿之类的命令
 - static/ 统计 api 例如: 资源总量查询之类的命令
 - creep/ 爬虫行为拓展**
 - function/ 普通方法/行为拓展
 - mission/ 任务相关 【后面会有详细说明】 [mission 是拼写错误,发现时已经不敢改了,后面同理]
 - move/ 寻路移动相关
 - position/ 位置拓展
 - function/ 普通方法拓展
 - powercreep/ pc 行为拓展
 - mission/ 任务相关 【后面会有详细说明】
 - move/ 寻路相关
 - function.ts 普通行为拓展
 - room/ 房间拓展 【重要】**
 - core/ 房间运行内核
 - init.ts 初始化
 - spawn.ts 孵化系统
 - ecosystem.ts 生态系统 (自动布局、修复、房间状态(战争、和平))
 - function/ 普通房间行为
 - mission/ 任务相关 【后面会有详细说明】
 - structure/ 建筑拓展
- main.ts 主函数
- utils.ts 杂项功能函数 各种杂七杂八的函数

主函数

```
export const loop = ErrorMapper.wrapLoop(() =>{  
  /* Memory初始化 */  
  MemoryInit()          // Memory room creep flag  
  
  /* 跨shard初始化 */  
  InitShardMemory()  
  
  /* 跨shard记忆运行 */  
  InterShardRun()  
  
  /* 原型拓展挂载 */  
  Mount()  
  
  /* 爬虫数量统计及死亡Memory回收 */  
  CreepNumStatistic()  
  
  /* 房间框架运行 */  
  RoomWork()  
  
  /* 爬虫运行 */  
  CreepWork()  
  
  /* 四人小队模块 */  
  SquadManager()  
  
  /* 资源调度超时管理 */  
  ResourceDispatchTick()  
  
  /* 像素 */  
  pixel()  
  
  /* 布局可视化 */  
  layoutVisual()  
  
})
```

下面根据主函数介绍整个框架的大致运行过程，其中，四人小队、跨 shard 等不介绍，只挑选重要的讲

整个框架的大致运行过程【只挑重要步骤】

1. 初始化 Memory

如下图所示，例如如果没有 Memory.whitesheet 就创建一个



2. 挂载原型拓展【这个没啥好说的】

3. 删除 G 了的爬虫的 Memory 以及统计每个房间，每个角色的爬虫的数量，这将在下一步孵化时用到。

4. 运行 RoomControlData 里的所有房间的逻辑，其中，每个房间的运行流程如下：

```
if (!Memory.RoomControlData) Memory.RoomControlData = {}
for (var roomName in Memory.RoomControlData)
{
    let thisRoom = Game.rooms[roomName]
    if (!thisRoom) continue
    /* 房间核心 */
    thisRoom.RoomInit()           // 房间数据初始化
    thisRoom.RoomEcosphere()      // 房间状态、布局
    thisRoom.SpawnMain()          // 常驻爬虫的孵化管理 [不涉及任务相关爬虫的孵化]

    /* 房间运维 */
    thisRoom.MissionManager()     // 任务管理器

    thisRoom.SpawnExecution()     // 孵化爬虫

    thisRoom.TowerWork()          // 防御塔工作

    thisRoom.StructureMission()   // terminal link factory 工作

    ResourceDispatch(thisRoom)    // 资源调度

    thisRoom.LevelMessageUpdate() // 房间等级Memory信息更新
}
```

首先是房间 memory 的初始化，这一点很重要，会更新很多很多房间的信息，例如各个建筑、矿的 id，其他 key 的初始化等。

房间状态、自动布局及自动修复，负责分辨房间处于何种状态[战争 or 和平]，检测到

房间升级时执行自动布局、检测到房间建筑缺损时进行自动修复

孵化管理，当然这只涉及房间的常驻爬虫，如：升级工、建筑工、采矿工等。任务相关爬虫的孵化管理在任务管理器里。

任务管理器 【重点】 顾名思义，负责管理任务的分发、销毁。当然也管理任务相关爬虫的孵化管理。这里只简单介绍，后续会详细介绍。

孵化函数 与孵化管理不同，孵化管理是负责把孵化信息推送到孵化队列里，孵化函数则是根据孵化队列里的信息孵化爬虫并更新孵化队列

防御塔工作函数 防御塔不使用任务管理，因为防御塔是实时的，任务调度有时差

建筑处理任务的函数 顾名思义

资源调度模块 每个房间处理资源调度任务的模块

房间等级 memory 更新，例如房间 7 升到 8 级了，memory 里的等级还是 7，需要更新一下。[配合其他函数，用于识别房间是否升级了]

5. 所有 pc 和爬虫运行相应逻辑

```
export default()=>{
  /* powercreep */
  for (var pc in Game.powerCreeps)
  {
    if (Game.powerCreeps[pc].ticksToLive)
    {
      Game.powerCreeps[pc].ManageMisson()
    }
  }

  /* creep */
  let adaption = true // 每tick执行一次adaption检查
  for (var c in Game.creeps)
  {
    let thisCreep = Game.creeps[c]
    if (!thisCreep) continue
    /* 跨shard找回记忆 */
    if (!thisCreep.memory.role) ...
  }
  if (!RoleData[thisCreep.memory.role]) continue
  // 自适应体型生产的爬虫执行恢复体型的相关逻辑
  if (adaption && thisCreep.memory.adaption && thisCreep.store.getUsedCapacity()==0 ) ...
  }
  /* 非任务类型爬虫 */
  let a = Game.cpu.getUsed()
  if (RoleData[thisCreep.memory.role].fun) ...
  }
  /* 任务类型爬虫 */
  else
  {
    thisCreep.ManageMisson()
  }
}
```

首先是 pc 的逻辑，ManageMisson()是 pc 执行任务的函数

普通爬虫的逻辑，包括跨 shard 找回记忆相关，还有任务类型爬虫和非任务类型爬虫的逻辑。

这里需要注意的是，非任务类型爬虫就是不执行任务，只执行其对应的函数的爬虫，它们不归任务调度。任务类型的爬虫由任务调度，领取任务、执行任务、操作任务。这里不会详细阐述，在后面的章节会详细阐述。

至此，主函数上最重要的模块的基础功能阐述完毕，整个代码的逻辑框架也差不多如此。接下来从各个模块阐述。

孵化系统

注：本手册不会具体讲孵化系统的源码，只讲其机理及如何使用。孵化系统我写的非常复杂，但是基本上可靠。再后来耦合到任务系统中，完全实现了不需要了解其底层机理即可进行开发。**一般来说，如果想使用并开发 sup bot 框架，只需要知道如何使用孵化系统就行了。不需要太了解源码。**

本框架的孵化从孵化方法上分为 2 类：**补员型孵化**（某类型爬虫数量低于指定数目，就添加孵化队列）和**间隔型孵化**（每隔一段时间添加一个孵化队列）。从孵化的管理上也分为 2 类：**常驻爬虫**（大部分是非任务型爬虫[上节有讲]，中央和物流搬运工是例外）的孵化管理和**任务爬虫**的孵化管理，他们由两个不同的函数分别管理。其中，**常驻爬虫不支持间隔型孵化**。当然，也有添加孵化队列的 api，完全可以根据自己喜好进行修改。

1. 爬虫信息常量

在 constant/SpawnConstant.ts 里

```
/* 爬虫信息列表 */
export const RoleData:SpawnConstantData = {
  'harvest':{num:0,ability:[1,1,2,0,0,0,0,0],adaption:true,level:5,mark:"^",init:true,fun:harvest_}, // 矿点采集工
  'carry':{num:0,ability:[0,3,3,0,0,0,0,0],level:5,mark:"📦",init:true,adaption:true,fun:carry_}, // 矿点搬运工
  'upgrade':{num:0,ability:[1,1,2,0,0,0,0,0],level:10,mark:"⬆️",init:true,fun:upgrade_}, // 升级工
  'build':{num:0,ability:[1,1,2,0,0,0,0,0],level:10,mark:"🏗️",init:true,fun:build_,must:true}, // 建筑工
  'manage':{num:0,ability:[0,1,1,0,0,0,0,0],level:2,mark:"👤",init:true,must:true,adaption:true}, // 中央搬运工
  'transport':{num:0,ability:[0,2,2,0,0,0,0,0],level:1,mark:"🚚",init:true,must:true,adaption:true}, // 房间物流搬运工
  'repair':{num:0,ability:[1,1,1,0,0,0,0,0],level:2,mark:"🔧",must:true}, // 刷墙
  'claim':{num:0,ability:[0,0,1,0,0,0,1,0],level:10,mark:"🐼"}, // 开房sf
  'cupgrade':{num:0,ability:[2,5,7,0,0,0,0,0],level:11,mark:"🐼"},
  'dismantle':{num:0,ability:[25,0,25,0,0,0,0,0],level:11,mark:"⚡"},
  'rush':{num:0,ability:[10,2,5,0,0,0,0,0],level:11,mark:"⬆️"},
  'truck':{num:0,ability:[0,10,10,0,0,0,0,0],level:9,mark:"🚚"},
  'claim':{num:0,ability:[0,0,1,0,0,0,1,0],level:10,mark:"🐼"},
  'Ebuild':{num:0,ability:[1,1,2,0,0,0,0,0],level:13,mark:"🏗️"},
  'defend-douHeal':{num:0,ability:[0,0,10,0,0,30,0,10],level:7,mark:"🟢",must:true},
  /* 四人小队 */
  'x-dismantle':{num:0,ability:[28,0,10,0,0,0,0,12],level:9,mark:"🟡",must:true,mem:{creepType:'attack'}},
  'x-heal':{num:0,ability:[0,0,10,0,2,26,0,12],level:9,mark:"🟢",must:true,mem:{creepType:'heal'}},
  'x-attack':{num:0,ability:[0,0,10,28,0,0,0,12],level:9,mark:"🔴",must:true,mem:{creepType:'attack'}},
  'x-range':{num:0,ability:[0,0,10,0,24,4,0,12],level:9,mark:"🔵",must:true,mem:{creepType:'attack'}},
  'x-aio':{num:0,ability:[0,0,10,0,10,20,0,10],level:9,mark:"🌈",must:true,mem:{creepType:'heal'}},
```

上图是部分爬虫信息列表，

key 是爬虫的角色名，

num 是数量，设置为 0（后续会自己调整），

ability:是体型，分别是[work,carry,move,attack,ranged_attack,heal,claim,tough]的数量，总数不要超过 50 即可

level:是孵化的优先级，默认为 10

mark:爬虫名字里的标志符（这是我个人喜好）

adaption:是否自适应（如果总能量不足，是否需要自适应体型，默认不会）

must:是否战争状态下也孵化（默认不会）

mem:爬虫出生时额外携带的 memory

fun:【重要】如果有 fun 就是爬虫只执行 fun 对应的函数，不接受任务调度

init:常驻爬虫需要 init 为 true，这样就归常驻爬虫的孵化管理器来管理

当然，细心的人会发现体型在这里是写死的，不用担心，我的框架允许不同房间等级决定动态体型，也允许自己决定体型，这将是底下需要讲到的。

下图是不同角色根据不同房间等级决定的体型和数量。举个例子，如果爬虫信息里体型常量为 1 work 1 move，房间等级为 8 级，对应的等级体型表里 8 级的体型为 2 work 2 move，则会覆盖上面的体型。

```

/* 爬虫部件随房间等级变化的动态列表 */
export const RoleLevelData = {
  > 'harvest':{ ...
  },
  > 'carry':{ ...
  },
  'upgrade':{
    1:{bodypart:[1,1,2,0,0,0,0,0],num:4},
    2:{bodypart:[2,2,4,0,0,0,0,0],num:3},
    3:{bodypart:[3,3,6,0,0,0,0,0],num:3},
    4:{bodypart:[4,4,8,0,0,0,0,0],num:2},
    5:{bodypart:[4,4,8,0,0,0,0,0],num:2},
    6:{bodypart:[5,2,5,0,0,0,0,0],num:2},
    7:{bodypart:[10,2,10,0,0,0,0,0],num:2},
    8:{bodypart:[15,3,15,0,0,0,0,0],num:1},
  },
  > 'build':{ ...
  },
  > 'transport':{ ...
  },
  > 'manage':{ ...
  },
  > 'repair':{ ...
  },
  > 'dismantle':{ ...
  },
  'rush':{
    6:{bodypart:[17,1,9,0,0,0,0,0],num:0},
    7:{bodypart:[39,1,10,0,0,0,0,0],num:0},
  }
}

```

上图中，RoleLevelData 下的 key 是角色名，角色名下为房间等级，房间等级对应的对象里 bodypart 是体型，num 是不同等级下的数量（只对常驻孵化有效）

那如果我们需要自定义体型呢，其实也有解决办法，在 global.SpecialBodyData[房间名][角色名]里可以写相应体型，这是优先级最高的，孵化的时候会按照这个体型孵化。如下图 1 个实例：

```

// 体型
global.SpecialBodyData[this.name]['aio'] = GenerateAbility(0,0,10,0,6,23,0,11)
if ((Game.time - global.Ctime[this.name]) % 10) return

```

这样设置之后，aio(一体机)的体型将会变为 10 move 6 range 23 heal 和 11 tough

总结，爬虫体型的优先级 global.SpecialBodyData 里的数据 > RoleLevelData 里的数据 > RoleData 里的数据。

2. 常驻爬虫的孵化管理

常驻爬虫的孵化管理信息存在房间的 Memory.SpawnConfig 里

如下图

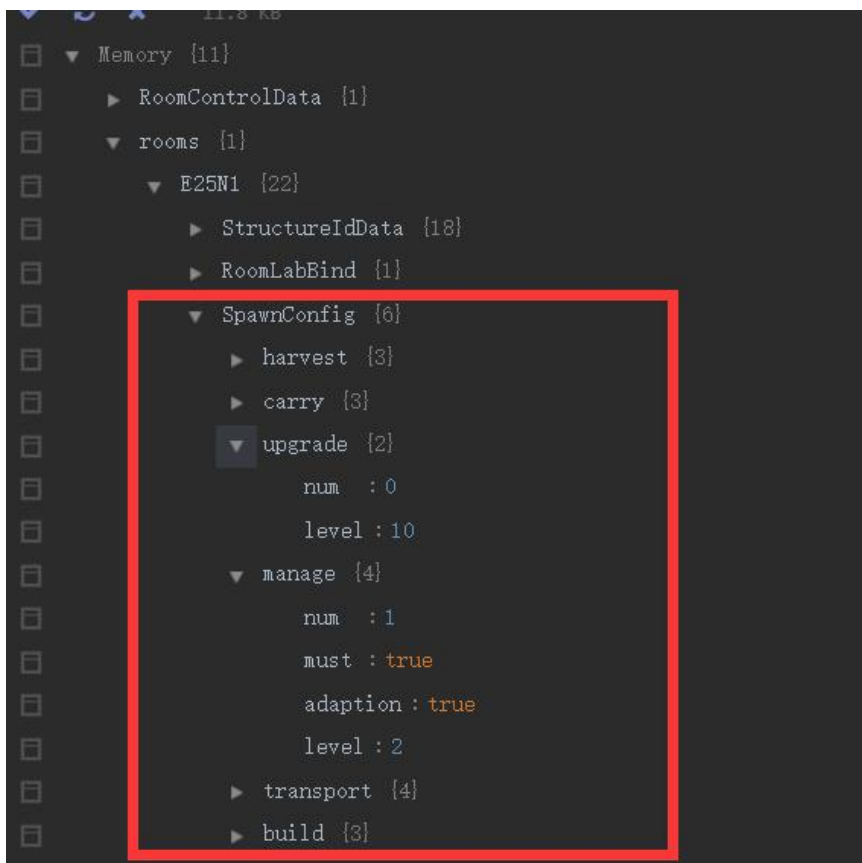
其对应的对象属性就是前面说的爬虫信息常量里的信息，num 是数量，体型不在此处。当每个房间对应的爬虫的数量低于 num 时，就会添加一个孵化命令进孵化队列。

这里有一些特殊说明：

每一次升级后，SpawnConfig 里的信息比如数量都会更新（根据 RoleLevelData 里的数量更新）

build 建造工，在 5 级前数量一直是 1，5 级后数量会自动变为 0，当检测到建筑工地时会变成 1

这些爬虫的数量可以手动灵活设置，有相应 api

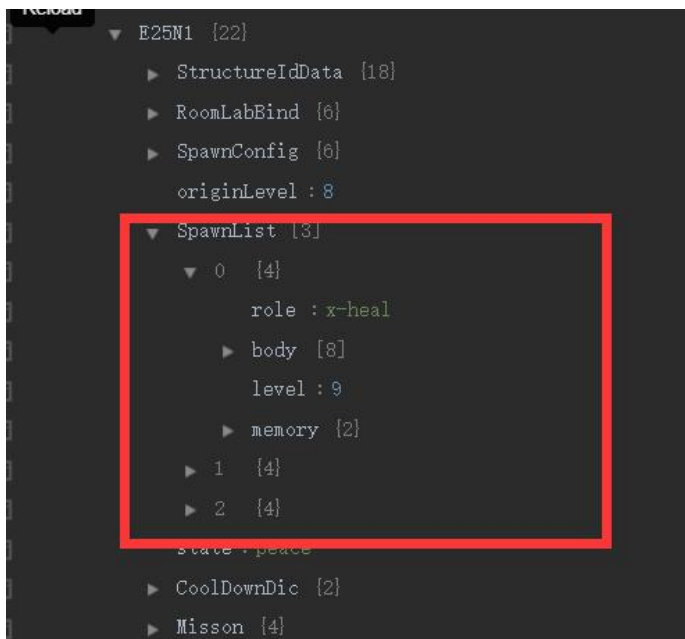


3. 任务爬虫孵化管理

因为还没有讲任务系统，这里简单说明

任务爬虫的孵化信息在任务对象里存储，包含数量、间隔时间（可选）等。

4. 孵化队列

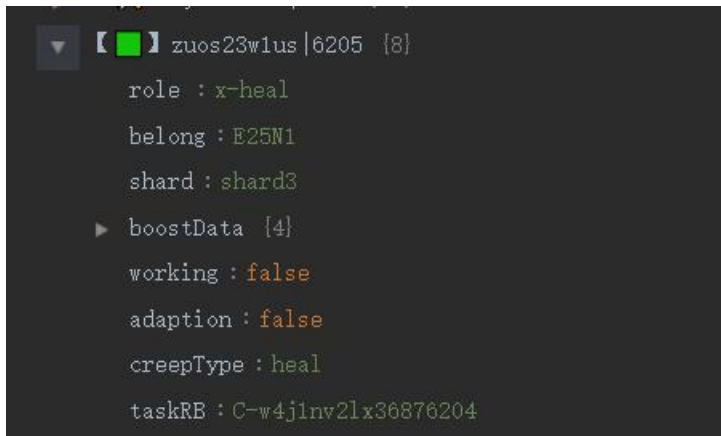


里面包含孵化信息包括：角色、体型、孵化优先级、希望携带的 **memory**

当然，不一定严格按照孵化队列里对象的体型孵化，如果能量不够且可以自适应，还是会自适应孵化的，或者 **global.SpecialBodyData** 里有数据，也会按照这个孵化。

5. 孵化

本框架里按照孵化对象里的优先级逐一进行孵化，孵化后的爬虫会自带一些信息



这是上图孵化队列里孵化后的爬虫一出生就自带的信息，非常重要

role: 爬虫角色

belong: 爬虫所属的房间

shard: 爬虫所属的 shard

boostData 爬虫的 boost 信息（如果需要 boost 需要用的到）

working: 简单的爬虫状态储存

adaption: 自动处理的属性，和自适应相关，无需处理

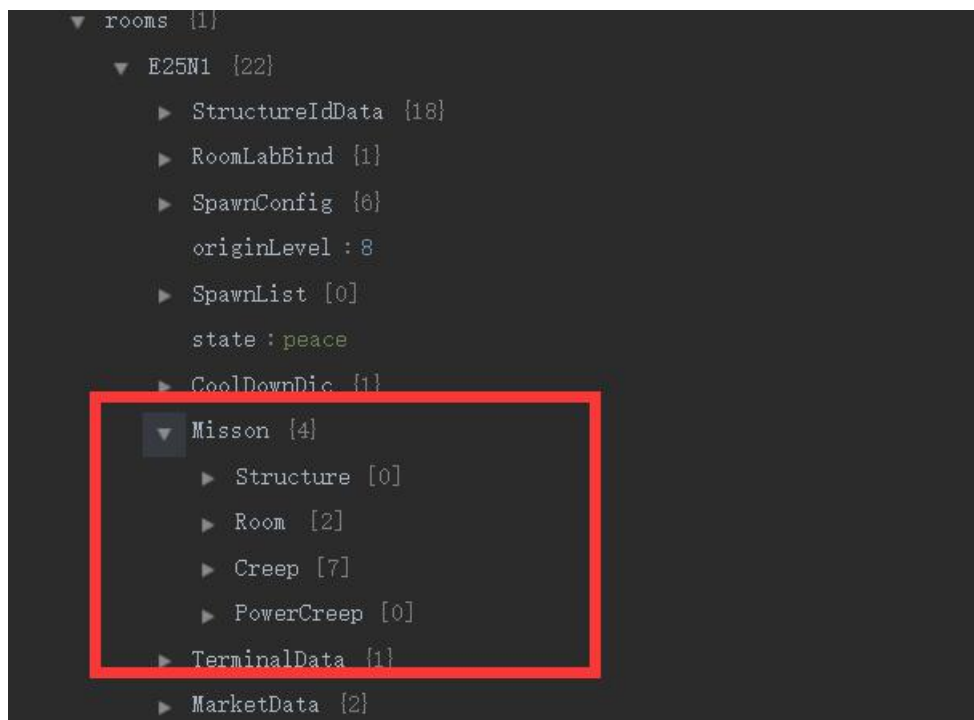
creepType: 四人小队相关的记忆 这属于孵化队列里的额外添加的记忆

taskRB: 这是间隔孵化爬虫都携带的属性，很重要，用来寻找自己对应的任务

以上，role belong shard boostData 是所有爬虫出生都有的属性，非常重要。

任务系统

任务系统是核心系统，调度房间中的绝大部分操作。【该部分看不懂的话，可以结合下面的一个实例看】 房间任务系统的原型拓展在 `mount/room/mission/` 里
每个任务都是一个**对象**，里面包含各个信息。



任务信息在房间.memory.Misson 里，存在四个分支，Structure[存储建筑相关任务]、Room[存储房间任务]、Creep[存储爬虫任务]、PowerCreep[存储 pc 的任务]

接下来让我们看看任务对象的定义，再分别做说明。

```
/* 房间任务模板 */
interface MissionModel{
  /* 所有任务都必须有 */
  name:string // 任务名称
  range: 'Room' | 'Creep' | 'Structure' | 'PowerCreep' // 任务所属范围 新增powerCreep任务
  delayTick:number // 过期时间 1000 99999 (x)
  structure?:string[] // 与任务有关的建筑id Structure A B(link) ['Aid']
  state?:number // 任务状态 0 (A 1) 1 (B 2 A 0) 2
  maxTime?:number // 最大重复任务数 默认1 例如我可以同时发布两个签名任务，1个去E1S1 一个去E1S2 物流运输( 3 )
  LabBind?:MissionLabBind // 实验室绑定 #
  cooldownTick?:number // 冷却时间 默认10 A -A cooldownTick= 10
  CreepBind?:BindData // 爬虫绑定 {'A':{num:1,bind:[],interval?:100}}
  level?:number // 任务等级，越小优先级越高 默认10
  Data?:any // 数据 {disRoom:xxxx,A:xxxx,B:xxx} Data ---> 浅拷贝 creep.memory.MissionData.Data
  reserve?:boolean // 适用于Creep范围的任务，即任务删除后，creepMemory里的任务数据不会删除 (默认会删除)
  /* 自动处理属性 */
  processing?:boolean // 任务是否正在被处理 只有在处理期间过期时间才会递减 ture --> delayTick -- false delayTick(x)
  id?:string // 每个任务的唯一标识 获取任务 删除任务 添加任务api id
}
```

name:任务的名称

范围: 顾名思义，决定该任务分配到 Room.memory.Misson 里的哪个分支里

delayTick: 任务超时时间，如果是 99999 就永远不会超时

structure: 执行任务的 structure 的 id 只适用于 structure 分支的任务

state: 任务状态

maxTime: 最大允许的重复任务数，默认为 1 表示该房间最多挂载几个同样 name 的任务

cooldownTick: 冷却时间，表示某一 name 的任务挂载成功后，过多久才允许再次挂载该类


型任务

CreepBind【重要，需要细讲】

只适合 Creep 类型任务，里面包含任务所需要的爬虫，以及其数量、孵化方式

在我的框架里，只需要定义 CreepBind 信息后，只要任务成功挂载并存在，就会自动执行孵化操作，这也是为什么我说孵化系统不需要详细去了解，只需要会用就行了。

下图为示例：



```
▼ 1 {9}
  name : 扩张援建
  range : Creep
  delayTick : 26661
  level : 10
  ► Data {1}
  reserve : true
  ▼ CreepBind {3}
    ▼ claim {2}
      num : 0
      ► bind [0]
    ▼ Ebuild {2}
      num : 2
      ▼ bind {2}
        0 : 【✖】6zh0mnrhng|6490
        1 : 【✖】jq1itdtoyj|6615
      ► Eupgrade {2}
  id : C-1446c290dj36863389
  processing : true
```

这是我房间里一个任务的 CreepBind 信息，当 bind 里的数量低于 num 时，并且房间内不存在没有领取任务的相应 role 的爬虫，就会进行孵化，当然，这是补员型孵化。还有个间隔型孵化，也差不多逻辑，有一点小区别，有兴趣的读者可以去了解。

也就是说，我们下需要 Creep 的任务时，只需要定义好需要的爬虫，孵化系统会自动帮我们处理。

顺便一提，爬虫的任务领取和任务中死去爬虫的信息更新，都由任务系统全自动处理，无需在意这些细节。

level: 默认 10 任务优先等级，假如两个任务，优先等级高的会排前面，优先被领取并执行。

Data: 任务中各种杂七杂八的数据，都可以放 Data 里。默认情况下，Data 里的数据在爬虫领取该任务后，也会拷贝一份进爬虫自己的记忆里。

reserve:在删除任务后，默认也会删除爬虫记忆里的相应任务数据，如果 reserve 为 true，就不会删除（这样即使任务删除了，爬虫也会继续执行他的任务）

id: 每个任务独一无二的 id，该 id 在挂载任务成功后，会自动获得，不需要手动处理

process:任务是否正在被处理,没有爬虫接取任务时默认 false,接取后为 true,当 process 为 true 时，任务的超时 tick 才会递减

多说无意，我们看看具体的任务对象

```
▼ 1 {9}
  name : 扩张援建
  range : Creep
  delayTick : 26661
  level : 10
  ▶ Data {1}
  reserve : true
  ▼ CreepBind {3}
    ▶ claim {2}
    ▼ Ebuild {2}
      num : 2
      ▼ bind {2}
        0 : 【❌】6zh0mnhng|6490
        1 : 【❌】jq1itdtoyj|6615
    ▶ Eupgrade {2}
  id : C-i446c290dj36863389
  processing : true
```

```
▼ 2 {8}
  name : deposit采集
  range : Creep
  delayTick : 2000
  level : 10
  ▶ Data {5}
  ▼ CreepBind {1}
    ▼ deposit {2}
      num : 1
      ▶ bind {0}
  maxTime : 0
  id : C-ihd7i6hwfp36876617
```

我们看这两个任务，第一个任务很明显 bind 里有爬虫，说明已经被接取了，process 为 true。他的 delayTick 也表示还有 26661tick 后任务就会过期，就会自动删除任务。第二个任务没有 process 说明还没被接取，此时，孵化队列已经有一个孵化 deposit 类型爬虫的队列了。每个任务都有 id,并且都不会一样
注：任务的 name 可以重复，但是 id 都是唯一的

任务管理器及任务 api 简单介绍

首先给出任务管理器的运行流程
如下图

```

4
5  /* 房间原型拓展  --任务  --任务框架 */
6  export default class RoomMissonFrameExtension extends Room {
7      /* 任务管理器 */
8      public MissionManager():void{
9          // 冷却监测
10         this.CoolDownCaculator()
11         // 超时监测
12         this.DelayCaculator()
13         // 任务-爬虫 绑定信息更新
14         this.UnbindMonitor()
15         // 任务-爬虫 孵化
16         this.MissonRoleSpawn()
17         /* PC任务管理器 */
18         this.PowerCreep_TaskManager()
19
20         /* [全自动] 任务挂载区域 需要按照任务重要程度进行排序 */
21         this.Spawn_Feed() // 虫卵填充任务
22         this.Task_CenterLink() // 能量采集
23         this.Task_ComsumeLink() // 消费、冲级link
24         this.Constru_Build() // 建筑任务
25         this.Task_clink() // 链接送仓任务
26         this.Tower_Feed() // 防御塔填充任务
27         this.Lab_Feed() // 实验室填充\回收任务
28         this.Nuker_Feed() // 核弹填充任务
29         this.Nuke_Defend() // 核弹防御
30         this.Task_CompoundDispatch() // 合成规划 (中级)
31         this.Task_monitorMineral() // 挖矿
32         this.Task_montitorPower() // 烧power任务监控
33         this.Task_Auto_Defend() // 主动防御任务发布
34
35         /* 基本任务监控区域 */
36         for (var index in this.memory.Misson)
37         for (var misson of this.memory.Misson[index])

```

注释里很清楚，冷却、超时、CreepBind 里爬虫信息更新（爬虫死亡后解绑任务之类）、任务爬虫孵化系统。

下面的都是具体的任务，注释里也很清楚。这里分为主动任务和被动任务，主动任务一般用来监测房间的信息，然后发布任务或者进行其他操作。被动任务则是被动处理的任务，它们往往需要被挂载才能执行。如下图所示

```

/* [全自动] 任务挂载区域 需要按照任务重要程度进行排序 */
this.Spawn_Feed() // 虫卵填充任务
this.Task_CenterLink() // 能量采集
this.Task_ComsumeLink() // 消费、冲级link
this.Constru_Build() // 建筑任务
this.Task_clink() // 链接送仓任务
this.Tower_Feed() // 防御塔填充任务
this.Lab_Feed() // 实验室填充\回收任务
this.Nuker_Feed() // 核弹填充任务
this.Nuke_Defend() // 核弹防御
this.Task_CompoundDispatch() // 合成规划 (中级)
this.Task_monitorMineral() // 挖矿
this.Task_montitorPower() // 烧power任务监控
this.Task_Auto_Defend() // 主动防御任务发布

/* 基本任务监控区域 */
for (var index in this.memory.Misson)
for (var misson of this.memory.Misson[index])
{
    A:
    switch (misson.name){
        case "物流运输":{this.Task_Carry(misson);break A;}
        case "墙体维护":{this.Task_Repair(misson);break A;}
        case "黄球拆迁":{this.Task_dismantle(misson);break A;}
        case "急速冲级":{this.Task_Quick_upgrade(misson);break A;}
        case "紧急援建":{this.Task_HelpBuild(misson);break A;}
        case "紧急支援":{this.Task_HelpDefend(misson);break A;}
        case "资源合成":{this.Task_Compound(misson);break A;}
        case "攻防一体":{this.Task_aio(misson);break A;}
        case "外矿开采":{this.Task_OutMine(misson);break A;}
        case "power升级":{this.Task_ProcessPower(misson);break A;}
        case "过道采集":{this.Task_Cross(misson);break A;}
        case "power采集":{this.Task_PowerHarvest(misson);break A;}
        case "红球防御":{this.Task_Red_Defend(misson);break A;}
        case "蓝球防御":{this.Task_Blue_Defend(misson);break A;}
        case "双人防御":{this.Task_Double_Defend(misson);break A;}
        case "四人小队":{this.Task_squad(misson);break A;}
    }
}

```

主动任务

被动任务

如何发布任务在以后会用一个完整的写任务的过程的实例来讲解，这里主要更多的是介绍。当然任务中也包含 lab 的绑定之类的信息，这个需要读者自己去看怎么实现的（其实如果是用框架或者在这个基础上开发的话，不需要详细了解）

有几个 api 在运行任务时很长用到，这里列举几个很常用到的 api

```
/* 添加任务 */
public AddMission(mis:MissionModel):boolean{ ...
}

/* 删除任务 */
public DeleteMission(id:string):boolean{ ...
}
```

添加任务和删除任务，添加任务时给出一个任务对象，直接添加即可
删除任务是给出 id，根据 id 删除任务

```
/* 任务数量查询 */
public MissionNum(range:string,name:string):number{ ...
}
/* 与role相关的任务数量查询 */
public RoleMissionNum(role:string,name:string):number{ ...
}

/* 获取任务 */
public GainMission(id:string):MissionModel | null{ ...
}

/* 通过名称获取唯一任务 */
public MissionName(range:string,name:string):MissionModel|null{ ...
}
```

这些判断任务数量、获取任务之类的 api 不详细介绍，自己根据需要使用，还有很多其他 api，在 Room/mission/base/base.ts 里有，根据自己需要看。

爬虫原型拓展：任务处理

根据上面对任务系统的介绍，假设我们下达任务了，那爬虫是怎么处理任务的呢？这里介绍的就是爬虫处理的函数。所有任务类型爬虫处理任务都是通过 `mount/creep/mission/base.ts` 里的 `ManageMission()` 处理，也就是所有任务类型的爬虫都会运行这个函数，前面讲的 `pc` 和爬虫运行逻辑也能发现。

```
public ManageMission():void{
    if (this.spawning) return
    if (!this.memory.MissionData) this.memory.MissionData = {}
    /* 生命低于10就将资源上交 */
    > if(this.ticksToLive < 10 && (isArray(['transport','manage'],this.memory.role))) ...
    }
    > if (Object.keys(this.memory.MissionData).length <= 0) ...
    }
    else
    {
        switch (this.memory.MissionData.name) {
            case '虫卵填充':{this.handle_feed();break;}
            case '物流运输':{this.handle_carry();break;}
            case '墙体维护':{this.handle_repair();break;}
            case 'C计划':{this.handle_planC();break;}
            case '黄球拆迁':{this.handle_dismantle();break;}
            case '急速冲级':{this.handle_quickRush();break;}
            case '扩张援建':{this.handle_expand();break;}
            case '紧急支援':{this.handle_support();break;}
            case '控制攻击':{this.handle_control();break;}
            case '紧急援建':{this.handle_helpBuild();break;}
            case '房间签名':{this.handle_sign();break;}
            case '攻防一体':{this.handle_aio();break;}
            case '原矿开采':{this.handle_mineral();break;}
            case '外矿开采':{this.handle_outmine();break;}
            case 'power采集':{this.handle_power();break;}
            case 'deposit采集':{this.handle_deposit();break;}
            case '红球防御':{this.handle_defend_attack();break;}
            case '蓝球防御':{this.handle_defend_range();break;}
            case '双人防御':{this.handle_defend_double();break;}
            case '四人小队':{this.handle_task_squad();break;}
        }
    }
}
```

基本的逻辑就是，任务类型爬虫 `memory` 里会有个 `MissionData` 的东西，这东西默认是 `{}` 空对象，如果为空爬虫就会寻找其对应房间有没有他的任务，如果有就领取任务，并把任务相关信息放进 `MissionData` 里，然后根据 `MissionData` 里的任务名执行相应任务操作。

领取了任务也会把任务对象的 `process` 设置为 `true`。代表该任务正在被处理了，这就是爬虫处理任务的逻辑了，非常简单可靠。

爬虫原型拓展：寻路

在 mount/creep/move/里有寻路的原型拓展，这是我和 E19N2(添加了过道优先策略)魔改的 hoho 的寻路版本。默认寻一次路，撞墙或者出现异常后会再次寻路。同时也支持对穿等操作。

用法也很简单：

目标在爬虫当前房间：

`creep.goTo(目标.pos,1)` 1 是范围，代表靠近目标 1 格就行了。

如果目标在其他房间，并且没有其他房间的视野，可以

`creep.goTo(new RoomPosition(24,24,房间名),24)`

如果有视野 `creep.goTo(目标.pos,1)`也行

总之，就是一句 `goTo` 走天下，所有房间都可以！

对于对穿：当 `creep.memory.standed` 为 `true` 时，则不允许其他爬虫对自己进行对穿

对穿时，对于 `creep.memory.crossLevel` 大于自己的对穿等级时，则不能对其进行对穿

给一些实例：

```
// 考虑到建筑和修复有可能造成堵虫，所以解除钉扎状态
public build_(distination:ConstructionSite) : void {
    if (this.build(distination) == ERR_NOT_IN_RANGE)
    {
        this.goTo(distination.pos,1)
        this.memory.standed = false
    }
    else
        this.memory.standed = true
}
```

这是建筑的便捷原型拓展函数，在建建筑的时候不允许被对穿，不修的时候允许。如果建筑不在自己这里，直接 `goTo` 过去（比 `moveTo` 省 cpu）

当然，如果想跨 shard 可以使用 `arriveTo(pos,范围,目标 shard)`

这个有一定局限性，代码我以后会修改。它不能去 `s0`。也就是说，你 `s3->s2`、`s2->s1`、`s1->s2` 等等都是可以的，但是不能 `s1->s0`。同时它会自动寻找最合适的十字路口。

紧急援建任务实例

接下来详细具体的给出创建一个紧急援建任务的开发步骤，完全理解这个步骤后，加上之前对整个我的代码框架的介绍，读者可以自己开发其他自己感兴趣的任務了

开发目标：

我需要写一个任务，爬虫需要能自动完成 **boost**，并且去目标房间进行援建，最好是间隔出爬。（**boost** 的话，涉及 lab 资源的搬运，这个我的代码里有一套成熟的机制，会自动发布搬运资源的任务）

开发步骤：

1. 定义角色

```
src > constant > TS SpawnConstant.ts > [0] RoleData
18  /* 爬虫信息列表 */
19  export const RoleData:SpawnConstantData = {
20    'harvest':{num:0,ability:[1,1,2,0,0,0,0,0],adaption:true,level:5,mark:"^",init:true,fun:harvest_},
21    'carry':{num:0,ability:[0,3,3,0,0,0,0,0],level:5,mark:"^",init:true,adaption:true,fun:carry_}, //
22    'upgrade':{num:0,ability:[1,1,2,0,0,0,0,0],level:10,mark:"^",init:true,fun:upgrade_}, // 升级工
23    'build':{num:0,ability:[1,1,2,0,0,0,0,0],level:10,mark:"^",init:true,fun:build_,must:true}, // 建
24    'manage':{num:0,ability:[0,1,1,0,0,0,0,0],level:2,mark:"^",init:true,must:true,adaption:true}, //
25    'transport':{num:0,ability:[0,2,2,0,0,0,0,0],level:1,mark:"^",init:true,must:true,adaption:true},
26    'repair':{num:0,ability:[1,1,1,0,0,0,0,0],level:2,mark:"^",must:true}, // 刷墙
27    'cclaim':{num:0,ability:[0,0,1,0,0,0,1,0],level:10,mark:"^",must:true}, // 开房sf
28    'cupgrade':{num:0,ability:[2,5,7,0,0,0,0,0],level:11,mark:"^",must:true},
29    'dismantle':{num:0,ability:[25,0,25,0,0,0,0,0],level:11,mark:"^",must:true},
30    'rush':{num:0,ability:[10,2,5,0,0,0,0,0],level:11,mark:"^",must:true},
31    'truck':{num:0,ability:[0,10,10,0,0,0,0,0],level:9,mark:"^",must:true},
32    'claim':{num:0,ability:[0,0,1,0,0,0,1,0],level:10,mark:"^",must:true},
33    'Ebuild':{num:0,ability:[1,1,2,0,0,0,0,0],level:13,mark:"^",must:true},
34    'Eupgrade':{num:0,ability:[1,1,2,0,0,0,0,0],level:13,mark:"^",must:true},
35    'double-attack':{num:0,ability:[0,0,10,28,0,0,0,12],level:10,mark:"^",must:true},
36    'double-heal':{num:0,ability:[0,0,10,0,0,28,0,12],level:10,mark:"^",must:true},
37    'claim-attack':{num:0,ability:[0,0,15,0,0,0,15,0],level:10,mark:"^",must:true},
38
39    'architect':{num:0,ability:[15,10,10,0,0,10,0,5],level:10,mark:"^",must:true},
40
41    'scout':{num:0,ability:[0,0,1,0,0,0,0,0],level:15,mark:"^",must:true},
42    'aio':{num:0,ability:[0,0,25,0,10,15,0,0],level:10,mark:"^",must:true},
43    'mineral':{num:0,ability:[15,15,15,0,0,0,0,0],level:11,mark:"^",must:true},
44    /* 外矿 */
45    'out-claim':{num:0,ability:[0,0,2,0,0,0,2,0],level:11,mark:"^",must:true},
```

如上图，我在 src/constant/SpawnConstant.ts 里 RoleData 里添加了个我的援建爬的角色

2. 定义任务对象

```
public Public_helpBuild(disRoom:string,num:number,shard?:string,time?:number):MissionModel{
  var thisTask:MissionModel = {
    name:'紧急援建',
    range:'Creep',
    delayRick:20000,
    level:10,
    Data:{
      disRoom:disRoom,
      num:num,
      shard:shard?shard:Game.shard.name
    },
    maxTime:2
  }
  thisTask.reserve = true
  thisTask.CreepBind = {
    'architect':{num:num,bind:[],interval:time?time:1000},
  }
  thisTask.LabBind = this.Bind_Lab(['XZH02','XLH20','XLH02','XGH02','XKH20'])
  if (thisTask.LabBind)
    return thisTask
  return null
}
```


我在 src\mount\room\mission\publish\publish.ts 里定义了一个房间的原型拓展函数，这个函数可以用来生成一个紧急援建的任务对象，里面的详细参数我都写上了，目标房间，目标 shard 之类的，包括援建爬的数量等等

我在 CreepBind 里定义了想要的爬，其中 interval 代表我希望这个爬能间隔一段时间出一波。如果没有 interval 就是补全式孵化，有的话就是间隔制孵化

还有个 LabBind，因为我需要用到 lab 来放 boost 和进行 boost 操作，我需要在任务对象里写好需要占用的 lab，当然这一切都由一个叫 Bind_lab 的函数自动执行，我们不需要做太多考虑，（如果 Bind_lab 返回 null 说明 lab 不够或者其他啥原因。）这样我们就完整实现了一个生成“紧急援建”的任务对象的函数。我又称之为**发布函数**

3. 定义完发布函数后，我们要在控制台上进行任务的发布和删除，怎么做到呢？于是，我在 src\mount\console\control\action.ts 里定义了这么一个对象，然后挂载到全局里，就可以在控制台上执行我的 api 了。

```
support:{
  // 紧急援建
  build(roomName:string,disRoom:string,num:number,interval:number,shard:shardName = Game.shard.name as shardName):string{
    var thisRoom = Game.rooms[roomName]
    if (!thisRoom) return `[support] 不存在房间${roomName}`
    let task = thisRoom.Public_helpBuild(disRoom,num,shard,interval)
    if (thisRoom.AddMission(task))
      return Colorful(`[support] 房间${roomName}挂载紧急援建任务成功 -> ${disRoom}`, 'green')
    return Colorful(`[support] 房间${roomName}挂载紧急援建任务失败 -> ${disRoom}`, 'red')
  },
  cbuild(roomName:string,disRoom:string,shard:shardName = Game.shard.name as shardName):string{
    var thisRoom = Game.rooms[roomName]
    if (!thisRoom) return `[support] 不存在房间${roomName}`
    for (var i of thisRoom.memory.Mission['Creep'])
    {
      if (i.name == '紧急援建' && i.Data.disRoom == disRoom && i.Data.shard == shard)
      {
        if (thisRoom.DeleteMission(i.id))
          return Colorful(`[support] 房间${roomName}紧急援建任务成功`, 'green')
        }
      }
    }
    return Colorful(`[support] 房间${roomName}紧急援建任务失败`, 'red')
  },
},
```

一个是发布援建任务，一个是删除任务。每个参数的意思不多介绍 interval 是孵化间隔时间 我们注意到里面用到里的 AddMission 和 DeleteMission 这些 api，他们返回 true 代表添加/删除任务成功, false 则是代表添加或者删除任务失败

4. 现在我们可以自由的添加和删除援建任务了，我们一添加，房间的 memory 里 Mission['Creep']里就会有援建的任务对象，删除则是能将其删除，但是我们现在的任务只是个空壳子，哪怕添加了不会执行任何操作。我们首先需要定义个房间处理这种任务的操作，在 src\mount\room\mission\action\vindicate.ts 里写了如下函数，并将其放到任务管理器里。

```
/* 紧急援建 */
public Task_HelpBuild(mission:MissionModel):void{
  if ((Game.time - global.Gtime[this.name]) % 9) return
  if (mission.LabBind)
  {
    if(!this.Check_Lab(mission,'transport','complex')) return // 如果目标lab的t3少于 1000 发布搬运任务
  }
}
```

第一行： 每格 9tick 执行一次

第二行：如果该任务有绑定的 lab，则使用 Check_lab 这个 api 来进行 boost 资源搬运的逻辑，它会自动下达搬运资源的任务，transport 是指让 transport 这个角色去搬，complex 是指从 terminal 和 storage 里搬资源。

当然，我的这里面集成了资源调度，如果房间内资源不够，会进行资源调度，调度不到还会自动买资源。

【一个命令就能执行这么多操作，这是很棒的对吧】

```
{
  A:
  switch (mission.name){
    case "物流运输":{this.Task_Carry(mission);break A;}
    case "墙体维护":{this.Task_Repair(mission);break A;}
    case "黄球拆迁":{this.Task_dismantle(mission);break A;}
    case "急速冲顶":{this.Task_Quick_upgrade(mission);break A;}
    case "紧急援建":{this.Task_HelpBuild(mission);break A;}
    case "紧急支援":{this.Task_HelpDefend(mission);break A;}
    case "资源合成":{this.Task_Compound(mission);break A;}
    case "攻防一体":{this.Task_aio(mission);break A;}
    case "外矿开采":{this.Task_OutMine(mission);break A;}
    case "power升级":{this.Task_ProcessPower(mission);break A;}
    case "过道采集":{this.Task_Cross(mission);break A;}
    case "power采集":{this.Task_PowerHarvest(mission);break A;}
    case "红球防御":{this.Task_Red_Defend(mission);break A;}
    case "蓝球防御":{this.Task_Blue_Defend(mission);break A;}
    case "双人防御":{this.Task_Double_Defend(mission);break A;}
    case "四人小队":{this.Task_squad(mission);break A;}
  }
}
```

上图是在任务管理器里注册一下。不注册的话不会执行。

5. 现在我们的房间检测到这个任务会执行搬 boost 资源到 lab 的操作了，相应角色爬虫也会自动孵化了，那么问题来了，我们的爬虫孵化后，毫无疑问它会自动领取这个任务，但是领取后他会干什么呢？这是个问题，需要我们去写爬虫的处理这个任务的逻辑，如下。

```
// 紧急援建
public handle_helpBuild():void{
  let missionData = this.memory.MissionData
  let id = missionData.id
  let data = missionData.Data
  if (!missionData) return
  if (this.room.name == this.memory.belong && Game.shard.name == this.memory.shard)
  {
    if (!this.BoostCheck(['move','work','heal','tough','carry'])) return
    if (this.store.getUsedCapacity('energy') <= 0)
    {
      let stroge_ = global.Stru[this.memory.belong]['storage'] as StructureStorage
      if (stroge_)
      {
        this.withdraw_(stroge_,'energy')
        return
      }
    }
  }
  if ((this.room.name != data.disRoom || Game.shard.name != data.shard) && !this.memory.swith)
  {
    this.heal(this)
    this.arriveTo(new RoomPosition(24,24,data.disRoom),23,data.shard)
  }
  else...
}
}
```

我在 src\mount\creep\misson\action.ts 里定义了个爬虫处理紧急援建任务的拓展函数。其中 missionData.Data 是浅拷贝了任务对象里 Data 里的数据，里面包含 shard 目标房间等信息。第一层判断，如果爬虫在自己的房间内，他会干啥？

如上代码所示：**BoostCheck!** 就是进行 boost，我们只需要在里面写上需要 boost 的部件，爬虫就会自己去 boost 了，这是全自动的。boost 完了他就会干其他事了。读者不需要详细了解其机制，爬虫会自动寻找对应的 lab 进行 boost，相关部件都 boost 了，该函数就会返回 true，就能进行下一步了！

然后我们再在爬虫处理任务的函数里注册一下，就彻底完成任务的书写了！

```
case '控制攻击':{this.handle_control();break}
case '紧急援建':{this.handle_helpBuild();break}
case '房间签名':{this.handle_sign();break}
```

src\mount\creep\misson\base.ts ManageMisson()

后言

上面大致介绍了我的框架的核心内容，并给出了个具体任务的实例，我相信通过这些足够了解如何在框架基础上进行开发了。

当然还有很多模块，比如跨 **shard**、资源调度等模块我没有详细介绍，因为这些不是重点，读者完全可以自己去看源码去理解。上面介绍的才是核心内容，其他的都是可有可无的。最主要的还是要阅读代码，这个开发笔记只是起辅助理解的作用。

superbitch bot 使用说明

1. 将房间名添加运行框架中

例：假设我有房间 E29S2，我想在这个房间上运行我的代码，只需要如下命令

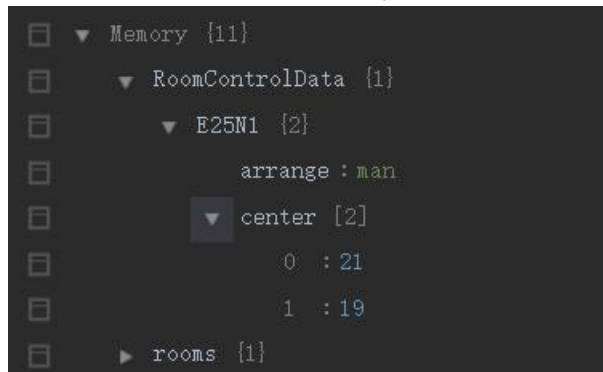
`frame.set("E25N1","man",24,21)`

第一个参数是房间名

第二个参数是布局 目前支持 dev 布局（参考 E45S49）和手动布局 man

后面两个参数是中心点的 x 和 y 坐标，这个很重要，后来会自动根据中心点决定近塔（修路的塔）和中央 link

设置好了后，你会发现 Memory 里多了这么个参数



2. 调整爬虫数量

例：假设我的升级工太少了，我想把升级工数量调整多一些

`spawn.num("E25N1","upgrade",10)`

第一个参数是房间名

第二个参数是我要调整的角色名，这个只支持常驻爬虫，也就是 upgrade harvest build carry transport manage 等角色，其中 manage 最多 1 个 transport 最多 2 个

第三个参数是数量，需要注意的是，每次房间升级后，该数量都会重置为默认，需要再次下命令更新数量

3. 删除房间中的建筑

例：我在房间 E25N1 中错放了个 road 建筑（无论是工地还是建筑），其坐标是 23,20

需要注意的是，不可以直接删，因为直接删的话，系统在检查到房间有建筑缺损后，会执行自动修复，所以需要使用 api 去删，命令如下

`frame.del("E25N1",23,20,"road")`

第一个参数是房间名

第二、三个参数是坐标

第四个参数是建筑的 structureType

4. 添加白名单、绕过房间

例：假设我的邻居 superbitch(E25N2)让我给他添加白名单，不然他就要揍我，我除了添加他进白名单，还需要让以后的爬寻路的时候绕过他的房间，防止擦枪走火。

`whitesheet.add("superbitch")`

`bypass.add("E25N2")`

第一个命令是添加白名单

第二个命令是所有的爬寻路都绕过他的房间

5. 修墙

例：长期遭受 superbitch 的欺压导致我对他有了戒备之心，我需要对我的房间修缮防御工事(ram 和 wall)

repair.set("E25N1",global,1,'LH')

第一个参数是房间名

第二个参数是修墙的类型 global 是房间内寻找 hit 最低的，special 是修插旗子的墙(还没写)，nuker 是防核弹修墙专用的（已经完成自动化防核，不需要手动下任务）

第三个参数是 boost 类型，不 boost 就是 null,boost 可选 LH LH2O XLH2O

当然删除修墙任务就是 **repair.remove("E25N1",global')**

6. 扩张

例：E25N1 一个房间不能满足我的野心，我想再扩张一个房间再 E29S2

expand.set("E25N1",E29S2,2,1)

第一个参数是房间名

第二个参数是目标房间

第三个参数是参与扩张的援建爬和升级爬数量

第四个参数是 claimer 的数量

扩张任务大概 30000tick 后会超时自动删除，手动删除命令

expand.remove("E25N1",E29S2)

7. 资源统计

例：我想查我总共有多少能量，我房间各个仓库是否饱和

resource.all()

store.all()

第一个命令查询资源

第二个命令查询容量信息

8. 任务输出屏蔽

例：我受够了控制台上那些任务挂载删除的提示，我决定屏蔽他们

MissionVisual.add("物流运输")

这样，物流运输挂载删除的提示就不会再出现在控制台里了

9. 资源传送

例：superbitch 问我索要 100K 的能量，不给就打我，我需要给他传送过去

terminal.send("E25N1",E25N2,'energy',100000)

第一个参数：我的房间名

第二个参数：目标房间名

第三个参数：资源类型

第四个参数：数量 不要超过 150K

10. 外矿

例：我想挖外矿了

mine.harvest("E25N1",10,20,"E24N1")

第一个参数：我的房间名

第三、三个参数：我的房间挖外矿的起始点（从这个点开始修路）

第四个参数：目标房间

取消则是 **mine.Charvest("E25N1","E24N1")**

11. 市场行为

```
/* 市场 */
market:{
  // 交易订单
  deal(roomName:string,id:string,amount:number):number{
    return Game.market.deal(id, amount, roomName);
  },
  // 查询订单
  look(rType:ResourceConstant,marType:"buy"|"sell"):string...
  },
  // 下买订单
  buy(roomName:string,rType:ResourceConstant,price:number,amount:number):string{ ...
  },
  // 查询平均价格
  ave(rType:ResourceConstant,day:number=1):string{
    return `[market] 资源${rType}在近${day}天内的平均价格为${ avePrice(rType,day)}`
  },
  // 查询是否有订单
  have(roomName:string,res:ResourceConstant,mtype:"sell"|"buy",p:number=null,r:num
  },
  // 查询市场上的最高价格
  highest(rType:ResourceConstant,mtype:'sell'|"buy",mprice:number=0):string{ ...
  },
  // 卖资源
  sell(roomName:string,rType:ResourceConstant,mType:'deal'|"order',num:number,pric
  },
  // 查询正在卖的资源
  query(roomName:string):string{ ...
  },
  // 取消卖资源
  cancel(roomName:string,mtype:'order'|"deal',rType:ResourceConstant):string{ ...
  },
},
```

太多了，自己摸索吧

12. 资源合成

例：我想合成 t3 去揍 superbitch

lab.init("E25N1")

lab.dispatch("E25N1",XLHO2,20000)

第一个命令是初始化房间的 lab(如果此前初始化过了，可以不用再初始化)

第二个命令是进行合成控制，系统会自动从 LO OH LHO2 XLHO2 一步一步合成到 XLHO2

参数分别为 房间名 目标化合物 数量

13. 烧 power

例：我想烧 power 了

power.switch("E25N1")

返回 true 代表开始烧，返回 false 代表关闭

14. 挖过道

例：我想挖 power 和 deposit

`cross.init("E25N1",['E24N0','E25N0','E26N0'])` // 初始化一下过道信息

`cross.add("E25N1",'E27N0')` // 添加过道房

`cross.remove("E25N1",'E27N0')` // 删除过道房

`cross.show("E25N1")` // 展示过道任务状态

```
> cross.show("E25N1")
< [cross] 房间E25N1的过道采集任务详情配置如下：
    房间：E23N0,E23S0,E24N0,E24S0,E25S0,E25N0,E26S0,E26N0
    power:false
    deposit:true
    目前存在如下任务：deposit采集任务 E25N1-->E24N0 state:1
```

`cross.power("E25N1")` // 开启、关闭挖 power （初始化后默认关闭）

`cross.deposit("E25N1")` // 开启、关闭挖 deposit （初始化后默认关闭）

15. 核弹

例：我想核平了 superbitch 房间 E25N2 的坐标为 10, 10 的 storage

`nuke,launch("E25N1",'E25N2',10,10)`

16. 打架

例：我想派一体机去骚扰 superbitch

`war.aio("E25N1",'E25N2',1,'shard3',1000,true)`

房间名 目标房间名 一体机数量 目标房间所在 shard 孵化间隔 是否 boost

取消则是 `war.Caio("E25N1",'E25N2','shard3')`

例：我想派四人小队去揍 superbitch

`war.squad("E25N1",E25N2,'shard3','A',1000)`

房间名 目标房间名 目标房间所在 shard 四人小队的配置 孵化间隔

【配置】

A: 2 红 2 绿

R: 2 蓝 2 绿

D: 2 黄 2 绿

DR: 1 黄 1 蓝 2 绿

DA: 1 黄 1 红 2 绿

RA: 1 蓝 1 红 2 绿

Aio: 4 一体机

取消任务: `war.Csquad("E25N1",E25N2,'shard3','A')`

例：我想派紫球去戳 superbitch 的房间

`war.control("E25N1",'E25N2',1000,'shard3')`

房间名 目标房间名 孵化间隔 目标房间所在 shard

取消任务: `war.Ccontrol("E25N1",E25N2,'shard3')`

例：我想派大黄去拆 superbitch

`war.dismantle("E25N1",'E25N2',1,false,1000)`

房间名 目标房间 数量 boost?(false) 孵化间隔
// 这个很老的命令了，还不支持跨 shard 和 boost 建议读者自己修改任务，参考一体机等任务

例：我的好朋友 ray 又挨打了，我要去帮他

`whitesheet.add("RayAdais")`

`war.support("E25N1","E26N2','double')`

17. 搬运任务

例：我要从 RoomPosition(12,12,'E25N2')上的仓库搬运 300K 的 energy 到 RoomPosition(34,22,'E25N1')上的仓库/终端,E25N1 孵化 10 个爬虫去搬

`carry.special("E25N1","energy',new RoomPosition(12,12,'E25N2'),new RoomPosition(34,22,'E25N1'),10,300000)`

取消任务 `carry.Cspecial("E25N1")`

18. 签名

`scout.sign("E25N1","E25N2','shard3','hello world')`