起步构建

本章来自己开发一个 Promise 实现，提升异步编程的能力。

首先声明定义类并声明 Promise 状态与值，有以下几个细节需要注意。

- executor 为执行者
- 当执行者出现异常时触发**拒绝**状态
- 使用静态属性保存状态值
- 状态只能改变一次，所以在 resolve 与 reject 添加条件判断
- 因为

resolve或

rejected方法在 executor 中调用，作用域也是 executor 作用域，这会造成 this 指向 window，现在我们使用的是 class 定义，this 为 undefined。

```
class HD {
  static PENDING = "pending";
  static FULFILLED = "fulfilled";
  static REJECTED = "rejected";
  (executor) {
    this.status = HD.PENDING;
    this.value = null;
    try {
      executor(this.resolve.bind(this), this.reject.bind(this));
    } catch (error) {
      this.reject(error);
    }
  }
  resolve(value) {
    if (this.status == HD.PENDING) {
      this.status = HD.FULFILLED;
      this.value = value;
    }
  }
  reject(value) {
    if (this.status == HD.PENDING) {
      this.status = HD.REJECTED;
      this.value = value;
    }
  }
```

```
}

```

下面测试一下状态改变

```
<script src="HD.js"></script>
<script>
  let p = new HD((resolve, reject) => {
    resolve("后盾人");
  });
  console.log(p);
</script>

```

#THEN

现在添加 then 方法来处理状态的改变，有以下几点说明

1. then 可以有两个参数，即成功和错误时的回调函数
2. then 的函数参数都不是必须的，所以需要设置默认值为函数，用于处理当没有传递时情况
3. 当执行 then 传递的函数发生异常时，统一交给 onRejected 来处理错误

#基础构建

```
then(onFulfilled, onRejected) {
  if (typeof onFulfilled != "function") {
    onFulfilled = value => value;
  }
  if (typeof onRejected != "function") {
    onRejected = value => value;
  }
  if (this.status == HD.FULFILLED) {
    try {
      onFulfilled(this.value);
    } catch (error) {
      onRejected(error);
    }
  }
  if (this.status == HD.REJECTED) {
```

```
    try {
      onRejected(this.value);
    } catch (error) {
      onRejected(error);
    }
  }
}

```

下面来测试 then 方法的，结果正常输出后盾人

```
let p = new HD((resolve, reject) => {
  resolve("后盾人");
}).then(
  value => {
    console.log(value);
  },
  reason => {
    console.log(reason);
  }
);
console.log("houdunren.com");

```

#异步任务

但上面的代码产生的 Promise 并不是异步的，使用 setTimeout 来将 onFulfilled 与 onRejected 做为异步宏任务执行

```
then(onFulfilled, onRejected) {
  if (typeof onFulfilled != "function") {
    onFulfilled = value => value;
  }
  if (typeof onRejected != "function") {
    onRejected = value => value;
  }
  if (this.status == HD.FULFILLED) {
    setTimeout(() => {
      try {
```

```
        onFulfilled(this.value);
      } catch (error) {
        onRejected(error);
      }
    });
  }
  if (this.status == HD.REJECTED) {
    setTimeout(() => {
      try {
        onRejected(this.value);
      } catch (error) {
        onRejected(error);
      }
    });
  }
}

```

现在再执行代码，已经有异步效果了，先输出了`houdunren.com`

```
let p = new HD((resolve, reject) => {
  resolve("后盾人");
}).then(
  value => {
    console.log(value);
  },
  reason => {
    console.log(reason);
  }
);
console.log("houdunren.com");

```

#PENDING 状态

目前 then 方法无法处理 promise 为 pending 时的状态

```
...
let p = new HD((resolve, reject) => {
```

```
  setTimeout(() => {
    resolve("后盾人");
  });
})
...

```

为了处理以下情况，需要进行几点改动

1. 在构造函数中添加 callbacks 来保存 pending 状态时处理函数，当状态改变时循环调用

```
(executor) {
 ...
  this.callbacks = [];
  ...
}

```

2. 将 then 方法的回调函数添加到 callbacks 数组中，用于异步执行

```
then(onFulfilled, onRejected) {
  if (typeof onFulfilled != "function") {
    onFulfilled = value => value;
  }
  if (typeof onRejected != "function") {
    onRejected = value => value;
  }
 if (this.status == HD.PENDING) {
    this.callbacks.push({
      onFulfilled: value => {
        try {
          onFulfilled(value);
        } catch (error) {
          onRejected(error);
        }
      },
      onRejected: value => {
        try {
```

```
          onRejected(value);
        } catch (error) {
          onRejected(error);
        }
      }
    });
  }
  ...
}

```

3. resovle 与 reject 中添加处理 callback 方法的代码

```
resolve(value) {
  if (this.status == HD.PENDING) {
    this.status = HD.FULFILLED;
    this.value = value;
    this.callbacks.map(callback => {
      callback.onFulfilled(value);
    });
  }
}
reject(value) {
  if (this.status == HD.PENDING) {
    this.status = HD.REJECTED;
    this.value = value;
    this.callbacks.map(callback => {
      callback.onRejected(value);
    });
  }
}

```

#PENDING 异步

执行以下代码发现并不是异步操作，应该先输出 `大叔视频` 然后是`后盾人

```
let p = new HD((resolve, reject) => {
```

```
  setTimeout(() => {
    resolve("后盾人");
    console.log("大叔视频");
  });
}).then(
  value => {
    console.log(value);
  },
  reason => {
    console.log(reason);
  }
);

```

解决以上问题，只需要将 resolve 与 reject 执行通过 setTimeout 定义为异步任务

```
resolve(value) {
  if (this.status == HD.PENDING) {
        this.status = HD.FULFILLED;
                this.value = value;
    setTimeout(() => {
      this.callbacks.map(callback => {
        callback.onFulfilled(value);
      });
    });
  }
}
reject(value) {
  if (this.status == HD.PENDING) {
        this.status = HD.REJECTED;
    this.value = value;
    setTimeout(() => {
      this.callbacks.map(callback => {
        callback.onRejected(value);
      });
    });
  }
}

```

#链式操作

Promise 中的 then 是链式调用执行的，所以 then 也要返回 Promise 才能实现

1. then 的 onReject 函数是对前面 Promise 的 rejected 的处理
2. 但该 Promise 返回状态要为 fulfilled，所以在调用 onRejected 后改变当前 promise 为 fulfilled 状态

```
then(onFulfilled, onRejected) {
  if (typeof onFulfilled != "function") {
    onFulfilled = value => value;
  }
  if (typeof onRejected != "function") {
    onRejected = value => value;
  }
  return new HD((resolve, reject) => {
    if (this.status == HD.PENDING) {
      this.callbacks.push({
        onFulfilled: value => {
          try {
            let result = onFulfilled(value);
            resolve(result);
          } catch (error) {
            reject(error);
          }
        },
        onRejected: value => {
          try {
            let result = onRejected(value);
            resolve(result);
          } catch (error) {
            reject(error);
          }
        }
      });
    }
    if (this.status == HD.FULFILLED) {
      setTimeout(() => {
        try {
          let result = onFulfilled(this.value);
```

```
          resolve(result);
        } catch (error) {
          reject(error);
        }
      });
    }
    if (this.status == HD.REJECTED) {
      setTimeout(() => {
        try {
          let result = onRejected(this.value);
          resolve(result);
        } catch (error) {
          reject(error);
        }
      });
    }
  });
}

```

下面执行测试后，链式操作已经有效了

```
let p = new HD((resolve, reject) => {
  resolve("后盾人");
  console.log("hdcms.com");
})
.then(
  value => {
    console.log(value);
    return "大叔视频";
  },
  reason => {
    console.log(reason);
  }
)
.then(
  value => {
    console.log(value);
  },
```

```
  reason => {
    console.log(reason);
  }
);
console.log("houdunren.com");

```

#返回类型

如果 then 返回的是 Promise 呢？所以我们需要判断分别处理返回值为 Promise 与普通值的情况

#基本实现

下面来实现不同类型不同处理机制

```
then(onFulfilled, onRejected) {
  if (typeof onFulfilled != "function") {
    onFulfilled = value => value;
  }
  if (typeof onRejected != "function") {
    onRejected = value => value;
  }
  return new HD((resolve, reject) => {
    if (this.status == HD.PENDING) {
      this.callbacks.push({
        onFulfilled: value => {
          try {
            let result = onFulfilled(value);
            if (result instanceof HD) {
              result.then(resolve, reject);
            } else {
              resolve(result);
            }
          } catch (error) {
            reject(error);
          }
        },
        onRejected: value => {
          try {
            let result = onRejected(value);
```

```
            if (result instanceof HD) {
              result.then(resolve, reject);
            } else {
              resolve(result);
            }
          } catch (error) {
            reject(error);
          }
        }
      });
    }
    if (this.status == HD.FULFILLED) {
      setTimeout(() => {
        try {
          let result = onFulfilled(this.value);
          if (result instanceof HD) {
            result.then(resolve, reject);
          } else {
            resolve(result);
          }
        } catch (error) {
          reject(error);
        }
      });
    }
    if (this.status == HD.REJECTED) {
      setTimeout(() => {
        try {
          let result = onRejected(this.value);
          if (result instanceof HD) {
            result.then(resolve, reject);
          } else {
            resolve(result);
          }
        } catch (error) {
          reject(error);
        }
      });
    }
  });
```

```
}

```

#代码复用

现在发现 pendding、fulfilled、rejected 状态的代码非常相似，所以可以提取出方法 Parse 来复用

```
then(onFulfilled, onRejected) {
  if (typeof onFulfilled != "function") {
    onFulfilled = value => value;
  }
  if (typeof onRejected != "function") {
    onRejected = value => value;
  }
  return new HD((resolve, reject) => {
    if (this.status == HD.PENDING) {
      this.callbacks.push({
        onFulfilled: value => {
          this.parse(onFulfilled(this.value), resolve, reject);
        },
        onRejected: value => {
          this.parse(onRejected(this.value), resolve, reject);
        }
      });
    }
    if (this.status == HD.FULFILLED) {
      setTimeout(() => {
        this.parse(onFulfilled(this.value), resolve, reject);
      });
    }
    if (this.status == HD.REJECTED) {
      setTimeout(() => {
        this.parse(onRejected(this.value), resolve, reject);
      });
    }
  });
}
parse(result, resolve, reject) {
  try {
```

```
    if (result instanceof HD) {
      result.then(resolve, reject);
    } else {
      resolve(result);
    }
  } catch (error) {
    reject(error);
  }
}

```

#返回约束

then 的返回的 promise 不能是 then 相同的 Promise，下面是原生 Promise 的示例将产生错误

```
let promise = new Promise(resolve => {
  setTimeout(() => {
    resolve("后盾人");
  });
});
let p = promise.then(value => {
  return p;
});

```

解决上面的问题来完善代码，添加当前 promise 做为 parse 的第一个参数与函数结果比对

```
then(onFulfilled, onRejected) {
  if (typeof onFulfilled != "function") {
    onFulfilled = value => value;
  }
  if (typeof onRejected != "function") {
    onRejected = value => value;
  }
  let promise = new HD((resolve, reject) => {
    if (this.status == HD.PENDING) {
      this.callbacks.push({
        onFulfilled: value => {
          this.parse(promise, onFulfilled(this.value), resolve, reject);
```

```
        },
        onRejected: value => {
          this.parse(promise, onRejected(this.value), resolve, reject);
        }
      });
    }
    if (this.status == HD.FULFILLED) {
      setTimeout(() => {
        this.parse(promise, onFulfilled(this.value), resolve, reject);
      });
    }
    if (this.status == HD.REJECTED) {
      setTimeout(() => {
        this.parse(promise, onRejected(this.value), resolve, reject);
      });
    }
  });
  return promise;
}
parse(promise, result, resolve, reject) {
  if (promise == result) {
    throw new TypeError("Chaining cycle detected for promise");
  }
  try {
    if (result instanceof HD) {
      result.then(resolve, reject);
    } else {
      resolve(result);
    }
  } catch (error) {
    reject(error);
  }
}

```

现在进行测试也可以得到原生一样效果了

```
let p = new HD((resolve, reject) => {
  resolve("后盾人");
```

```
});
p = p.then(value => {
  return p;
});

```

#RESOLVE

下面来实现 Promise 的 resolve 方法

```
static resolve(value) {
  return new HD((resolve, reject) => {
    if (value instanceof HD) {
      value.then(resolve, reject);
    } else {
      resolve(value);
    }
  });
}

```

使用普通值的测试

```
HD.resolve("后盾人").then(value => {
  console.log(value);
});

```

使用状态为 fulfilled 的 promise 值测试

```
HD.resolve(
  new HD(resolve => {
    resolve("houdunren.com");
  })
).then(value => {
  console.log(value);
});

```

使用状态为 rejected 的 Promise 测试

```
HD.resolve(
  new HD((_, reject) => {
    reject("reacted");
  })
).then(
  value => {
    console.log(value);
  },
  reason => {
    console.log(reason);
  }
);

```

#REJEDCT

下面定义 Promise 的 rejecte 方法

```
static reject(reason) {
  return new HD((_, reject) => {
    reject(reason);
  });
}

```

使用测试

```
HD.reject("rejected").then(null, reason => {
  console.log(reason);
});

```

#ALL

下面来实现 Promise 的 all 方法

```
static all(promises) {
  let resolves = [];
  return new HD((resolve, reject) => {
    promises.forEach((promise, index) => {
      promise.then(
        value => {
          resolves.push(value);
          if (resolves.length == promises.length) {
            resolve(resolves);
          }
        },
        reason => {
          reject(reason);
        }
      );
    });
  });
}

```

来对所有 Promise 状态为 fulfilled 的测试

```
let p1 = new HD((resolve, reject) => {
  resolve("后盾人");
});
let p2 = new HD((resolve, reject) => {
  reject("后盾人");
});
let promises = HD.all([p1, p2]).then(
  promises => {
    console.log(promises);
  },
  reason => {
    console.log(reason);
  }
);

```

使用我们写的 resolve 进行测试

```
let p1 = HD.resolve("后盾人");
let p2 = HD.resolve("houdunren.com");
let promises = HD.all([p1, p2]).then(
  promises => {
    console.log(promises);
  },
  reason => {
    console.log(reason);
  }
);

```

其中一个 Promise 为 rejected 时的效果

```
let p1 = HD.resolve("后盾人");
let p2 = HD.reject("rejected");
let promises = HD.all([p1, p2]).then(
  promises => {
    console.log(promises);
  },
  reason => {
    console.log(reason);
  }
);

```

#RACE

下面实现 Promise 的 race 方法

```
static race(promises) {
  return new HD((resolve, reject) => {
    promises.map(promise => {
      promise.then(value => {
        resolve(value);
      });
    });
  });
```

```
}

```

我们来进行测试

```
let p1 = HD.resolve("后盾人");
let p2 = HD.resolve("houdunren.com");
let promises = HD.race([p1, p2]).then(
  promises => {
    console.log(promises);
  },
  reason => {
    console.log(reason);
  }
);

```

使用延迟 Promise 后的效果

```
let p1 = new HD(resolve => {
  setInterval(() => {
    resolve("后盾人");
  }, 2000);
});
let p2 = new HD(resolve => {
  setInterval(() => {
    resolve("houdunren.com");
  }, 1000);
});
let promises = HD.race([p1, p2]).then(
  promises => {
    console.log(promises);
  },
  reason => {
    console.log(reason);
  }
);
```