

```
// Usa Promise.  
Promise.resolve();
```

---



# Por que programação assíncrona?

- não travar fluxo de execução
- não travar renderização
- multiprocessamento (JS é monothread, mas tem o event-loop)
- executar algo ou não, se o estado for alterado



# O que é assíncrono no JS?

- requisições (XMLHttpRequest, fetch)
- eventos
- timers (setTimeout, setInterval)



# Callback

um callback é uma **função**  
**passada a outra função como**  
**argumento**, que é então  
**invocado** dentro da função  
externa **para completar alguma**  
**ação**.

```
// Callback para todo login
ccid.d("on.login", function(u, err) {
    console.log(
        "hi, my name is:", u.name,
        "avatar:", u.avatar(25)
    );
});
```

```
// ou
function onLogin (u, err) {
    console.log(
        "hi, my name is:", u.name,
        "avatar:", u.avatar(25)
    );
}
ccid.d("on.login", onLogin);
```

**TypeScript** pode  
ajudar muito nas  
**assinaturas** das  
funções



# Callback | Problemas

- Código espalhado e fluxos confusos

```
const download = (url, callback: (data) => void) => {  
  const xhr = new XMLHttpRequest();  
  // ...  
  xhr.onload = () => {  
    const reader = new FileReader();  
    reader.onload = () => {  
      callback({  
        type: xhr.response.type,  
        base64: reader.result  
      });  
    };  
    reader.readAsDataURL(xhr.response);  
  };  
  xhr.send();  
}
```

```
const savePhoto = data => {  
  // ... dispara o save dos dados de data  
}  
  
const onClick = (e: MouseEvent) => {  
  e.preventDefault();  
  download(photo, savePhoto);  
}
```



# Callback | Problemas

- Precisa de flags de controle de fluxo

```
let data = null;
let isFetching = false;

function getData(param, cb) {
  if (isFetching) {
    setTimeout(() => getData(param, cb), 500);
  }

  if (data !== null) {
    cb(null, data);
    return;
  }

  isFetching = true;
  fetchData(param, (response) => {
    isFetching = false;
    data = response;
    cb(null, data);
  });
}
```



# Promises

uma Promise ("*promessa*") representa a eventual **conclusão** ou **falha** de uma operação assíncrona e o seu valor resultante

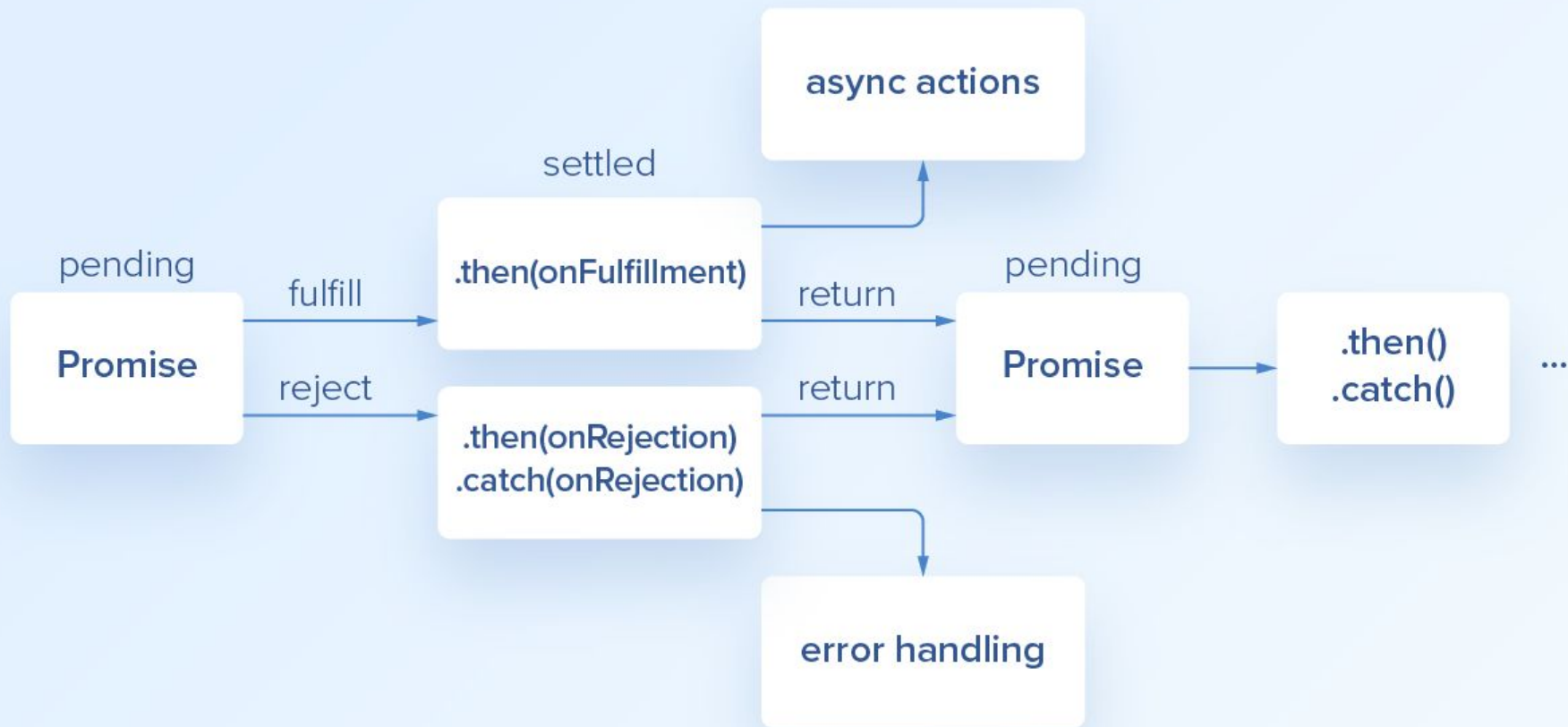
```
const getAlbumLetras = (artist, album) =>
  fetchJsonp(`https://api.letras.mus.br/v3/albums/${artist}/${album}?json=1`)
    .then(response => response.json())
    .catch(e => console.error("parsing failed", e));
```



# Promises | Estados

- pending (pendente): estado inicial, que não foi realizada nem rejeitada
- fulfilled (realizada): sucesso na operação
- rejected (rejeitado): falha na operação
- settled (estabelecida): que foi realizada ou rejeitado







## Promises | .then()


adiciona os métodos de tratamento  
da conclusão da Promise e retorna  
uma nova Promise

## Promises | .catch()

trata rejeição para a Promise e retorna  
uma nova Promise

```
let timer;  
const searchArtists = value => {  
  clearTimeout(timer);  
  return new Promise(resolve => {  
    timer = window.setTimeout(() => {  
      fetch("endpoint")  
        .then(data => data.json())  
        .then(locations => locations.map(/* ... */))  
        .then(results => resolve(results))  
        .catch((err) => console.error("Eita...", err));  
    }, 500);  
  });  
};
```

```
searchArtists("skank")  
  .then(artists => console.log(artists));
```



# Resolver, ou rejeitar, uma Promise não para o fluxo de execução dentro da Promise

```
const onAuth = ({ email, password }) =>  
  new Promise((resolve, reject) => {  
    ccid.d(  
      "auth",  
      (user, err) => {  
        if (err) {  
          reject(err);  
        }  
        console.log("fez login!!!");  
        resolve(user);  
      },  
      { email, password }  
    );  
  });
```

```
onAuth({ email:"teste" , password: "R3n41$san<3" })  
  .then(user => console.log("user", user))  
  .catch(e => console.error("error", e));
```

```
//> fez login!!!  
//> error {code: "u01", msg: "E-mail ou senha incorreto."}
```



# Promises | Vantagens

- fluxo encadeado
- API para tratamento de sucesso ou erro bem definida
- é reutilizável
- tratamento de coleções



# Promises | O que não fazer | pt1

```
const getUser = () =>
  new Promise<CCID.User>((resolve, reject) =>
    window.ccid.d("get.user", (user, err) => (err ? reject(err) : resolve(user)))
  );

if (getUser().then(user => user.isLogged())) {
  // ... do something
}
```



**.then()** retorna outra Promise, e isso é *truthy*



# Promises | O que não fazer | pt1 - sugestão

```
const getUser = () =>
  new Promise<CCID.User>((resolve, reject) =>
    window.ccid.d("get.user", (user, err) => (err ? reject(err) : resolve(user)))
  );

getUser().then(user => {
  if (user.isLogged()) {
    // ... do something
  }
}).catch(err => console.error(err));
```



# Promises | O que não fazer | pt2

```
class User extends React.Component {  
  render() {  
    <div>  
      <h1>  
        {fetch("http://get-user.com/ID" )  
          .then(res => res.json())  
          .then(user => user.name) }  
      </h1>  
    </div>  
  }  
}
```



# Promises | O que não fazer | pt2 - sugestão

```
class User extends React.Component {
  state = { name: "" };

  componentDidMount() {
    fetch("http://get-user.com/ID")
      .then(res => res.json())
      .then(user => {
        this.setState({ name: user.name });
      });
  }

  render() {
    <div>
      <h1>{this.state.name || "Carregando..."}</h1>
    </div>
  }
}
```



# Promises | Casos de reject | catch => undefined

```
const searchLetras = text =>
  fetch(`${URL}/letras/app/?q=${text}&limit=6`)
    .then(response => response.json())
    .catch(err => {
      console.error("parsing failed", err);
    });
```

```
searchLetras("switchfoot")
  .then(data => console.log("then", data))
  .catch(err => console.error("catch", err));
```

```
//> parsing failed ...
//> then undefined
```

! catch sem retorno !



# Promises | Casos de reject | tratar catch de fora

```
const searchLetras = text =>
  fetch(`${SolrCDN}/letras/app/?q=${text}&limit=6`)
    .then(response => response.json())

searchLetras("switchfoot")
  .then(data => console.log("then", data))
  .catch(err => {
    console.error("parsing failed", err);
  });
```



# Promises | Casos de reject | retorno no catch

```
const searchLetras = text =>
  fetch(`${SolrCDN}/letras/app/?q=${text}&limit=6`)
    .then(response => response.json())
    .catch(err => {
      console.error("parsing failed", err);
      return {};
    });

searchLetras("switchfoot").then(data => console.log("then", data));

//> parsing failed ...
//> then {}
```



# Promises | Casos de reject | reject no then

```
const searchLetras = text =>
  fetch(`${SolrCDN}/letras/app/?q=${text}&limit=6`)
    .then(response => response.json())
    .then(
      data => data,
      err => {
        console.error("parsing failed", err);
        return {};
      }
    );

searchLetras("switchfoot").then(data => console.log("then", data));

//> parsing failed ...
//> then {}
```



# Promises | API

- `.all()`
- `.race()`
- `.resolve()`
- `.reject()`



# Promises | .all(iterable)

método utilizado quando várias tarefas assíncronas podem ser executadas independentemente, mas é necessário que todas estejam prontas para prosseguir com o fluxo da aplicação.

```
const buildPromises = [buildCSS(), buildJS()];

Promise.all(buildPromises)
  .then(() => cleanSourceFiles())
  .catch((err) => emitError(err));
```



## Promises | .race(iterable)

método utilizado quando o sucesso ou erro de uma tarefa assíncrona numa lista de tarefas é suficiente para prosseguir com o fluxo da aplicação. Um bom caso de uso é quando se espera que apenas uma das promises seja *fulfilled* e as outras dêem *reject* em caso de alguma exceção que atrapalhe o resultado da primeira.



# Promises | .race(iterable)

```
const rejectOnRouteChange = () => new Promise((resolve, reject) => {
  router.onChangeRoute(() => reject("ROUTE_CHANGED"));
});

Promise.race(fetch("http://jquery-is-not-a-lang.com/api"), rejectOnRouteChange())
  .then(data => {
    renderData(data);
  }).catch(e => {
    if (e !== "ROUTE_CHANGED") {
      throw new Error(e);
    }
  });
```





# Promises | .resolve(value) e .reject(value)

métodos utilizados num contexto onde espera-se que uma promise seja retornada, mas o resultado da promise já foi definido pelo contexto e nenhuma tarefa assíncrona é necessária.

```
getUserScore() {  
  if (user.isReady()) {  
    return Promise.resolve(user.score);  
  }  
  return user.init().then(() => {  
    return user.score;  
  });  
}
```



# Não tem como cancelar o fluxo de execução de Promises e Callbacks

=/

<https://github.com/hjylewis/trashable>



# Funções assíncronas (com `async/await`)

- retornam Promise
- maneira mais limpa de escrever um fluxo com Promises dependentes
- as Promises rejeitadas nos *await* são como *reject* da função assíncrona



# Funções assíncronas (com async/await)

com Promise:

```
function getLocation() {  
  return new Promise(resolve => {  
    api.getMaster('geoip/city/').then(data => { // {"city": "Belo Horizonte", "region": 15}  
      // [{"cityId":176,"stateId":11 ...}]  
      api.get('cities', { name: data.city, region: data.region }).then(city => {  
        resolve(city[0]);  
      });  
    });  
  });  
}
```



# Funções assíncronas (com async/await)

com async/await:

```
async function getLocation() {  
  const geoipData = await api.getMaster("geoip/city/");  
  const city = await api.get("cities", { name: geoipData.city, region: geoipData.region });  
  return city[0];  
}
```

```
getLocation().catch((e) => console.error("Promise rejeitada no await com reason: ", e));
```

---

Dúvidas?



# Referências

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)
- <https://developers.google.com/web/fundamentals/primers/promises>
- <https://medium.com/trainingcenter/entendendo-promises-de-uma-vez-por-todas-32442ec725c2>